THE EXPERT'S VOICE® IN OPEN SOURCE

Beginning django CMS

Build a first-class content management system from the ground up the easy way

Nigel George

Foreword by Daniele Procida from Divio, the company behind django CMS

django CMS

Apress®

Beginning django CMS

Nigel George

Apress®

Beginning django CMS

ISBN-13 (pbk): 978-1-4842-1670-5

ISBN-13 (electronic): 978-1-4842-1669-9

Managing Director: Welmoed Spahr
Lead Editor: Ben Renow-Clarke
Development Editor: James Markham
Technical Reviewer: Massimo Nardone
Editorial Board: Steve Anglin, Pramila Balen, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing
Coordinating Editor: Melissa Maldonado
Copy Editor: James A. Compton
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springer.com`. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc. is a **Delaware** corporation.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

Any source code or other supplementary materials referenced by the author in this text is available to readers at `www.apress.com`. For detailed information about how to locate your book's source code, go to `www.apress.com/source-code/`.

To Kate, for always being there, and to my boys

—Chris and Zac.

You make me proud.

Contents at a Glance

Contents

About the Author



Nigel George is a business systems developer specializing in the application of Open Source technologies to solve common business problems. He has a broad range of experience in software development—from writing database apps for small businesses to developing the back end and UI for a distributed sensor network at the University of Newcastle, Australia.

Nigel also has over 15 years of experience in technical writing for business. He has written several training manuals and hundreds of technical procedures for corporations and Australian government departments. He has been using Django since version 0.96, and has programmed in C, C#, C++, VB, VBA, HTML, JavaScript, Python and PHP. He has also worked with frameworks/CMSes: WordPress, Django CMS, SharePoint, Joomla, Drupal, and Zope.

Nigel lives in Newcastle, NSW, Australia.

About the Technical Reviewer



Massimo Nardone is an experienced Android, Java, PHP, Python, and C++ programmer, technical reviewer, and expert. He holds a Master of Science degree in Computing Science from the University of Salerno, Italy. He worked as a PCI QSA and Senior Lead IT Security/Cloud/SCADA Architect for many years and currently works as Security, Cloud, and SCADA Lead IT Architect for Hewlett-Packard Finland. He has more than 20 years of work experience in IT, including the security, SCADA, cloud computing, IT infrastructure, mobile, security, and WWW technology areas for both national and international projects. Massimo has worked as a Project Manager, Cloud/SCADA Lead IT Architect, Software Engineer, Research Engineer, Chief Security Architect, and Software Specialist. He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He has been programming and teaching how to program with Perl, PHP, Java, VB, Python, and C/C++ for almost 20 years. He holds four international patents, in the PKI, SIP, SAML, and proxy areas.

Foreword

The world of Django was a very different place in September 2008, when Divio open-sourced django CMS.

Django itself had just had its long-awaited 1.0 release and was new and exciting. Django is now established and mature, and django CMS has developed along with it to become a recognized and dependable choice for web publishers.

Like Django, django CMS began life as an internal tool, and rapidly found new users when it was open-sourced. More than seven years later, finding that our software appeals to new users is still one of the most thrilling parts of the open source software adventure.

Beginning django CMS is a milestone in the story of django CMS. It represents a moment when the software starts to reach not just more new users, but a new kind of user altogether. That is, django CMS is becoming a realistic choice not only for the Django developers wanting to integrate a CMS into their projects, but also for people from beyond the usual world of Django and software developers—people who are more comfortable perhaps with a friendly book in their hands than with a README file in a GitHub repository.

You might be one of those people, in which case we extend you a special welcome. This book's for you. It takes a new approach into django CMS, one we heartily endorse, that doesn't make the assumptions about knowledge and experience that without doubt have undermined many people's attempts to get there.

For us, the community of django CMS developers, it represents an important challenge: to find ways to make the experience of newcomers to django CMS—people like you—a better one. Achieving this is essential to the long-term success of the project. We've been working toward it for some time, and Nigel's book is a very timely contribution to this effort.

It's an exciting moment for us, and it will be especially exciting to see the results: new django CMS users and the new websites they create.

So from us at Divio and in the django CMS development community, thanks to Nigel George and Apress for helping django CMS take another important step in its journey, and thank you, too, for joining us in it.

—Daniele Procida
Community & Documentation Manager, Divio AG, Zürich

Acknowledgments

I'd like to thank Ben, Melissa, Massimo, and the team at Apress. It's been a valuable learning experience, and the book is all the better for your input.

I would also like to thank Daniele Procida, Matteo Larghi, and Martin Koistinen from Divio AG for their support and assistance with some of the more technical details of django CMS; in particular, I would like to thank Daniele for writing such a fitting Foreword—your support is greatly appreciated.

Introduction

Beginning django CMS is for Internet developers who are sick and tired of dealing with complicated, bloated website frameworks that are a pain to build and a nightmare to maintain. django CMS is an open source website-building framework that is experiencing exponential growth because it is built on the simple, secure, and scalable architecture of Django. This book will take you from knowing potentially nothing about django CMS to building a functional website and content management system you can deploy for your own website or for your customers.

Who This Book Is For

Beginning django CMS is for developers who wish to build a custom content management system (CMS), simply and cleanly, without the overhead and bloat that comes with many other CMS frameworks.

To get the most out of this book, you will ideally have a basic knowledge of programming, web development, and Internet technologies. django CMS is built on Django, which is written in Python, and so a working knowledge of Python will make understanding many of the examples much easier.

Having said that, the book is not too technical for new developers to understand; the content has been written so that a new developer with minimal prior experience will be able to create a fully functioning django CMS website by the end of the book.

My only suggesting to those of you who are just starting out is that you familiarize yourself with all the free resources listed in Chapter 11; so if you do get stuck on some technical issue, you will know where to look for the answers.

What You'll Learn

In this book, you will learn how to:

- Install and configure django CMS.
- Create and manage content in django CMS.
- Easily create custom site and page templates for visually stunning websites that work perfectly on all devices—from mobile phones to desktop computers.
- Create and extend navigation menus for your django CMS website.

- Install and use a range of plugins from django CMS and third-party developers.
- Install and integrate other Django applications with your django CMS website.
- Create your own custom plugins and Django applications that integrate with django CMS.

For a complete chapter summary, as well as a more detailed outline of the book, see Chapter 1.

Software Versions

This book is written for django CMS version 3.2.

You are required to have Python installed before installing django CMS. At the time of writing, django CMS version 3.2 supports Python 2.6.x or later and Python 3.3.x or later. However, all of the code in this book is written in Python 3, so it's recommended that you install Python 3.4.3 or later. See Chapter 2 for details.

Once Python is installed, if you follow the installation instructions in Chapter 2, the django CMS installer will install all other software dependencies for you, including the latest supported version of Django.

If you wish to manually install django CMS, or need a custom configuration, see `http://docs.django-cms.org/en/latest/how_to/install.html`.

Source Code

All of the source code for this book can be downloaded from Apress. For detailed information about how to locate the source code, go to `www.apress.com/source-code/`.

CHAPTER 1

Working with a CMS

In this first chapter we'll be taking a high-level view of what a content management system (CMS) is and introducing django CMS: a modern content management system based on the Django Framework.

We will start with a very brief history of content management and how digital versions of old paper-based systems have evolved to handle the vast quantities of information that is disseminated across the world every minute of the day.

I will close the chapter with an outline of the contents of this book and a summary of the things you will learn as you go.

A Brief History

Back in the old days of the Internet (10 years ago), there was a strong move away from static HTML websites to more dynamic, interactive websites. This was great for users, but it presented a particular set of challenges to web designers. To meet the challenge, most professional websites were built in a particular way:

- A *designer* would create the overall design of a website.
- The designer would work with a *developer* to slice the design up into a site template that would be a mix of plain HTML and program code; with the program code providing the interactive functionality of the website.
- The designer would then work with a *content creator* (usually the client) to incorporate the website content into the template—duplicating the template and creating the content for each and every page of the website.

On very small sites, these roles could be filled by a single person, but for larger sites whole teams of designers, developers, and content creators needed to be involved to produce a complex website.

This design approach has one major flaw—the admixture of code, content, and design elements means that any change to one element has a very real chance of affecting the others. The consequences of this can range from simple formatting errors to major breakdowns of website functionality.

The solution to this problem was a *Content Management System (CMS)*—a set of methodologies and processes that have been around since long before the Internet.

Content Management 101

Content management is the systematic management of information (content). A simple way to understand content management is to use an example—this book:

- The *writer* (me) will give my draft to an *editor* who will work with me to ensure the manuscript is ready to go to the publisher.
- The *publisher* will approve the final manuscript for publication.
- The publisher will then send the properly laid out manuscript to a *printer* who produces the book you are now holding in your hand.

Of course, the actual process in more complex and much more collaborative than this sequence would suggest, but the important thing to note is that each person in the process fulfils a role and can function independent of the others; a publisher does not have to know the writer's craft, nor does the writer have to understand how to print a book.

Content Management Roles

A website CMS requires roles similar to those required to produce a book:

- *Authors* create the articles and page content that is displayed on the website.
- *Editors* take author-created content and edit it to fit the style and context of the website, as well as make any corrections to spelling and grammar. The editor usually fulfils the publisher role on a website CMS; marking content ready for displaying to the public via the CMS software running on the server.
- The *designer* still has an important role in creating the overall visual presentation of the website, although there is an increasing move towards buying off-the-shelf templates that suit a website owner's design goals, rather than employing a designer to create a custom look.
- The *site administrator* looks after the back end of the website—assigning roles to other users, maintaining the content, and applying backups and updates. In some cases, the site administrator is also the developer—writing code to improve the functionality of the website—although this is becoming less common, as you will see in the next section.

Anatomy of a Modern CMS

In almost all cases, a modern CMS is a computer application that can be desktop or cloud-based (or a combination of both). They allow for easy organization of content and provide tools for creating, editing and publishing content. They use database and file system storage to provide flexible and scalable storage of data; a modern CMS is capable of handling everything from a two-page website to massive content publishing platforms with millions of pages of content (like online news).

The vast majority of modern websites are running some form of CMS. WordPress, the oldest and most popular, runs roughly 1 in 5 websites on the Internet. CMS software has grown exponentially in popularity and functionality in recent years.

The current state of the art in CMS design combines all of the content management tools we have discussed with three design philosophies to create feature rich, scalable, and responsive websites:

- **Model/View/Controller (MVC) architecture**. It is beyond the scope of this book to explain MVC, but it is a software design pattern that epitomizes the modular approach. In MVC the content, the design, and the code are separated physically in the website, not just conceptually.
- **Use of external applications to increase functionality**. Rather than write new code, a modern CMS allows you to "plug in" additional programs written by other developers that provide new functions, rather than having to write the code yourself. These programs can range from simple plugins that change the formatting to full-blown applications. Using external applications not only increases the reliability of the CMS, but it also dramatically reduces the cost, as one developer can sell the same plugin or application to many different websites that need the functionality.
- **Mobile-friendliness**. Use of smartphones and tablet computers exceeds desktop use in most countries around the world. It is predicted that the shift to mobile computing will continue to increase over the next few years. It is a given that a modern CMS must support mobile devices by providing what is called a *responsive interface* that automatically adjusts the content of the website to suit the device it is being viewed on.

django CMS has some key advantages over other CMSes—most notably providing a dramatic improvement in modularity and separation of functions over programs like WordPress, which still use the old model of mixing content and code on the page. Other advantages include these:

- It is multilingual.
- It supports multiple websites.
- It uses Bootstrap out of the box to support mobile devices.

- It can connect to multiple databases.
- Its flexible page model and the availability of formatting plugins free you from rigid template layouts.
- It offers content scheduling out of the box.
- It provides clear separation of publishing and administration roles.
- It can integrate with other Django applications easily without major modifications.
- Its plugin-based design allows easy integration of email and third-party applications and content services.

django CMS combines all of these design elements into a clean, simple interface that is a delight to use for content creators, editors, and administrators. django CMS is built upon Django—a web content management system that was originally developed for a large US newspaper. It was released as open source software in 2007 and has continued to build a strong user base thanks to its robust architecture and scalability.

What You Will Learn

With django CMS you can build feature-rich, bug-free websites easily and quickly. This book will take you through the learning process; concluding with a basic blog website that you can use for your own blog, or to showcase your skills to an employer.

The aim of this book is to teach you the design methodology of django CMS and how to build a functioning website with it quickly and professionally; it's not designed to give you an in-depth, technical knowledge of django CMS. For deeper understanding and more advanced topics, I have provided resources and links in Chapter 11.

To give you a taste of what is to come:

- In Chapter 2 you will learn how to install django CMS and its dependencies on your computer.
- In Chapter 3, I will provide an overview of django CMS covering its design philosophy and a high-level view of the interface and its functions.
- In Chapter 4 you will learn about django CMS templates and how they leverage Django's template engine and Bootstrap to create fast, clean, and mobile-friendly websites.
- In Chapter 5 we will build a full set of responsive templates for your website. I will take you through each line of code so you will have a thorough grasp of how to build your own.
- In Chapters 6 and 7 we will be exploring the key tools that give django CMS its flexibility and power: plugins. We will cover all of the built-in plugins as well as some important third-party plugins that are available for django CMS.

- In Chapter 8 you will learn about organizing and managing content with django CMS. We will look at user management, user roles, and the publishing cycle within django CMS. At the end of Chapter 8 you will complete an exercise where you create the content for your website.
- In Chapter 9, you will learn about menus and navigation in django CMS. You will learn how to add breadcrumb navigation to your site, install social media sharing capability, and generate site maps. I will also show you how to create a custom menu.
- In Chapter 10, we will dig into some of the more advanced features of django CMS: extending the Page and Title models, extending the toolbar, and creating custom plugins. We will finish with a custom plugin that you can use to create a right sidebar menu for your site.
- Finally, in Chapter 11, I will give you tips and links to resources to continue your journey with django CMS. There are a lot of things django CMS is capable of, as well as some advanced topics that I did not have the scope to cover in this book. Chapter 11 will help you take those next steps to mastering django CMS.

Summary

In this chapter, we have briefly explored the history of content management and how a modern computer-based content management system (CMS) functions. We have looked at how django CMS fits into this ecosystem and highlighted some of its key advantages.

I closed with a brief outline of the book and what you can expect by the end. In the next chapter you will get underway with django CMS by installing the django CMS application and its dependencies.

CHAPTER 2

Installing django CMS

django CMS is an open source web-based content management system based on the Django web framework. It is maintained by Divio AG, a commercial web development company based in Zurich. But before you can start learning how to use django CMS, you must first install some software on your computer. Fortunately, it is a simple three step process:

1. Install Python.
2. Install a Python Virtual Environment.
3. Install django CMS.

If this does not sound familiar to you, don't worry; in this chapter I assume you have never installed software from the command line before and will lead you through it step by step.

While the user base for Mac OSX and Linux is growing all the time, I have written this section for the majority of you who will be using Windows. Your computer can be running any recent version of the Windows operating system (Windows Vista, Windows 7, Windows 8.1, or Windows 10).

The process is very similar on machines running Mac OSX and Linux. If you are using Mac or Linux, don't worry, as there are a large number of resources on the Internet with instructions on installing Python and django CMS on these systems. For additional information on installing software on these systems, see Chapter 11.

Note Windows users, depending on how your computer has been configured, you will most likely need to be running in a user account with administrator rights to be able to install software and run some commands and the command prompt.

Installing Python

Installing Python is very simple, but first you need to check whether Python is already installed on your system. To do this, we use a tool that may not be familiar to some Windows users— the command prompt. If you are comfortable using the command prompt, feel free to skim through this section.

Check Whether Python Is Installed

The command prompt is actually quite useful. As you learn django CMS you will use it quite a bit, so now is a good time to familiarize yourself with the tool if you have not used it before.

To launch the command prompt, you need to find `cmd.exe` and run the program.

In Windows 7 and Vista, click the Start button and type `cmd` into the Search Programs and Files box.

In Windows 8.1 and 10, type `cmd` into the Search the Web and Windows box.

In all versions of Windows, a list will appear—look for the file named `cmd.exe`. Click that file to run it, and you should be greeted by a black box similar to the one in Figure 2-1.

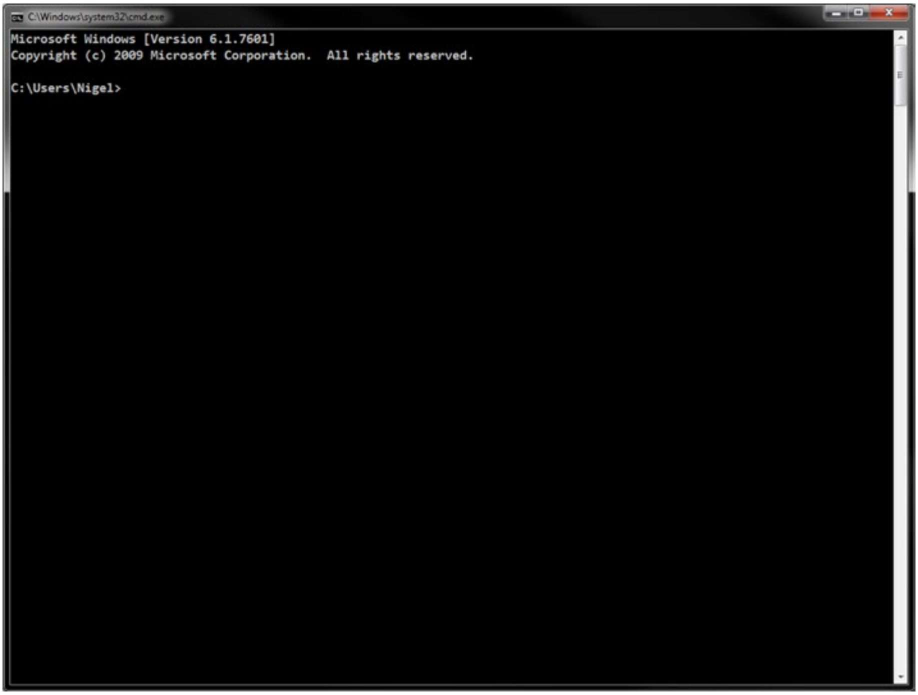


Figure 2-1. *The Windows command prompt*

■ **Tip** While you have the program running, right-click the cmd.exe icon in the taskbar (the black icon with c:\ _ in text) and select Pin to Taskbar. That way you will have easy access to the program each time you want to run it.

Now that we have the command window open, we need to check whether Python is already installed. This is easy—just type `python` at the prompt and hit Enter:

```
C:\Users\YourUsername>python
```

If Python is not installed, you will get a message like this:

```
'python' is not recognized as an internal or external command,
Operable program or batch file.
```

If Python is installed, you will get a message similar to the one shown in Figure 2-2, and it will open the Python interactive prompt. We will be getting back to Python's interactive prompt shortly.

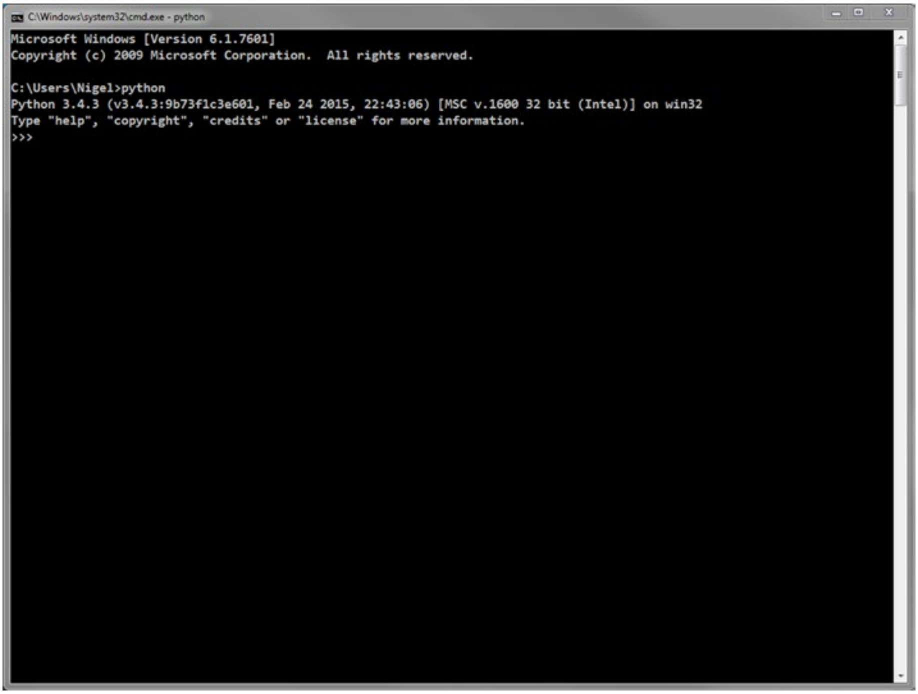


Figure 2-2. *The Python interactive prompt*

■ **Warning** If the Python version listed by your system starts with a 2 (such as 2.7.x), you are running an older version of Python and will need to install Python 3 as all the code in this book is written in Python 3. If you need to install Python 3, please follow the instructions in the next section.

Close the command window when you are done.

Download and Install Python

Assuming Python is not installed in your system, we first need to get the installer. Go to `https://www.python.org/downloads/` and click the big yellow button that says Download Python 3.x.x.

At the time of writing, the latest version of Python is 3.4.3, but it may have been updated by the time you read this, so the numbers may be slightly different.

Do not download version 2.7.x, as this is the old version of Python. All of the code in this book is written in Python 3, so you will get compilation errors if you try to run the code on Python 2. If for some reason you must use Python 2 and not Python 3, you will need to modify some of the code. In most cases, this is as simple as changing `__str__()` functions back to `__unicode__()` and modifying the `print()` function back to the `print` statement; however, other errors may appear. For links to complete references on the differences between the two versions, please see Chapter 11.

Once you have downloaded the Python installer, go to your Downloads folder and double click the file `python-3.x.x.msi` to run the installer.

The installation process is the same as for any other Windows program, so if you have installed software before, there should be no problem here; however, there is one extremely important customization you must make.

■ **Caution** Do not forget this next step as it will solve most problems that arise from incorrect mapping of `pythonpath` (an important variable for Python installations) in Windows.

When the installer opens the customization window, the option "Add python.exe to Path" is not selected; you must change this to "Will be installed on local hard drive" as shown in Figure 2-3.

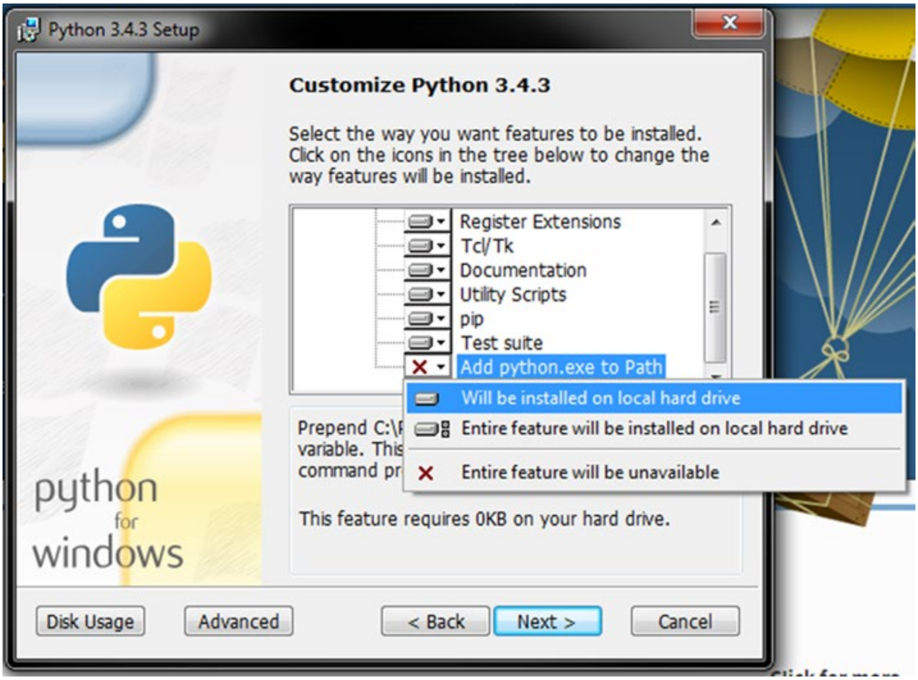


Figure 2-3. *The Add python.exe to Path option*

Once Python is installed, you should be able to reopen the command window and type `python` at the command prompt to get the same output as Figure 2-2. Do that now to make sure Python is installed and working.

While you are at it, there is one more important thing to do.

Exit out of Python by typing Ctrl-C. At the command prompt type the following and hit Enter:

```
C:\Users\YourUsername>python -m pip install -U pip
```

You don't need to understand exactly what this command does right now; put briefly, `pip` is the Python package manager. It is used to install Python packages: `pip` is actually a recursive acronym for "Pip Installs Packages." `Pip` is important for the next stage of our installation process, but first we need to make sure we are running the latest version of `pip`, which is exactly what this command does.

Installing a Python Virtual Environment

All of the software on your computer operates interdependently—each program has other bits of software that it depends on (called *dependencies*) and settings that it needs to find the files and other software it needs to run (called *environment variables*). When you are writing new software, it is possible (and common!) to modify dependencies and environment variables that your other software depends on. This can cause numerous problems, so it should be avoided.

■ **Note** The Python virtual environment, which allows you to run isolated instances of Python on your machine, should not be confused with a virtual machine. The latter is software that allows you to run entire operating systems and applications as if they were a physical computer.

A Python virtual environment solves this problem by wrapping all the dependencies and environment variables that your new software needs into a file system separate from the rest of the software on your computer. The virtual environment tool in Python is called `virtualenv`, and we install it from the command line using `pip` (Figure 2-4):

```
C:\Users\YourUsername>pip install virtualenv
```

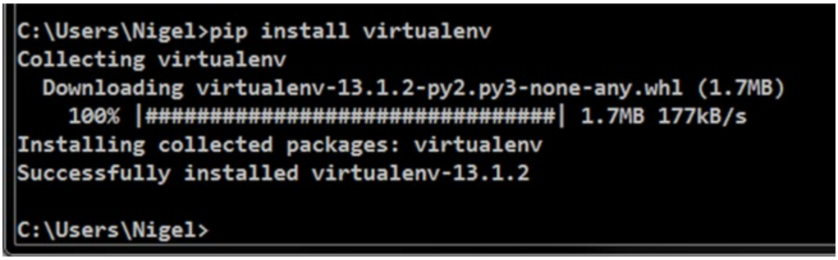


Figure 2-4. *Pip installed successfully*

Once `virtualenv` is installed, you need to create a virtual environment for your project (Figure 2-5):

```
C:\Users\YourUsername>virtualenv myCMS
```

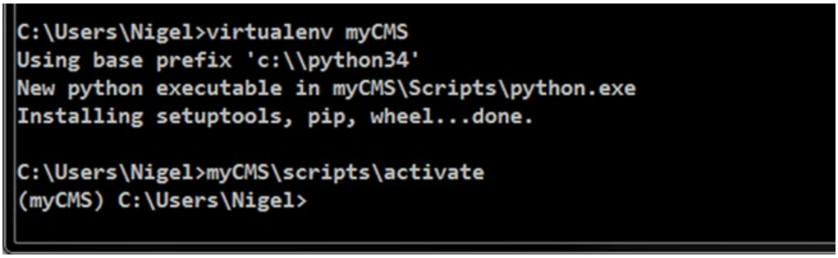


Figure 2-5. *Python* `virtualenv` *active*

In this case, I have called the new virtual environment `myCMS`, but you can call it whatever you like. Once `virtualenv` has finished setting up your new virtual environment, open Windows Explorer and have a look at what `virtualenv` created for you.

In your home directory, you will now see a folder called `\myCMS` (or whatever name you gave the virtual environment). If you open the folder, you will see the following:

```
\Include
\Lib
\Scripts
 \src
```

`Virtualenv` has created a complete Python installation for you, separate from your other software, so you can work on your project without affecting any of the other software on your system. To use this new Python virtual environment we have to activate it, so let's go back to the command prompt and type the following:

```
C:\Users\YourUsername>myCMS\scripts\activate
```

This will run the `activate` script inside your virtual environment's `\scripts` folder. You will notice that your command prompt has now changed:

```
(myCMS) C:\Users\YourUsername>
```

The `(myCMS)` at the beginning of the command prompt lets you know that you are running in the virtual environment. Our next step is to install our django CMS project.

Installing django CMS

The first step to install django CMS is to install the django CMS installer. This is done with `pip`:

```
(myCMS) C:\Users\YourUsername>pip install djangocms-installer
```

Then we have to create a project directory for our django CMS project and change into that directory:

```
(myCMS) C:\Users\YourUsername>mkdir myCMS-tutorial
(myCMS) C:\Users\YourUsername>cd myCMS-tutorial
```

Finally, we need to install the django CMS-powered website:

```
(myCMS) C:\Users\YourUsername\myCMS-tutorial>djangocms -p . mysite
```

Don't forget the "." between `-p` and `mysite`!

The django CMS installer will now ask you a few questions. In most cases, you just accept the default; however, there are some options that you should change. I have highlighted them in bold.

```
Database configuration (in URL format) [default sqlite://localhost/project.db]:
django CMS version (choices: 2.4, 3.0, 3.1, stable, develop) [default stable]:
Django version (choices: 1.4, 1.5, 1.6, 1.7, 1.8, stable) [default stable]:
Activate Django I18N / L10N setting (choices: yes, no) [default yes]:
Install and configure reversion support (choices: yes, no) [default yes]:
Languages to enable. Option can be provided multiple times, or as a comma
separated list. Only language codes supported by Django can be used here: en
Optional default time zone [default <your timezone>]:
Activate Django timezone support (choices: yes, no) [default yes]:
Activate CMS permission management (choices: yes, no) [default yes]:
Use Twitter Bootstrap Theme (choices: yes, no) [default no]: yes
Use custom template set [default no]:
Load a starting page with examples after installation (english language
only). Choose "no" if you use a custom template set. (choices: yes, no)
[default no]: yes
Creating the project
Please wait while I install dependencies

##lots more messages…

Creating admin user
Username (leave blank to use '<yourusername>'):
Email address: enter email address
Password: enter a password
Password (again):
Superuser created successfully.
All done!
```

Once django CMS is installed, you should be able to run your brand-new project. Inside your project directory (you should still be there if you launch straight after installing), you run the Django development server with the following command:

```
(myCMS) C:\Users\YourUsername\myCMS-tutorial>python manage.py runserver
```

Once the development server is running, you can enter `http://127.0.0.1:8000` into your browser and the default django CMS template will load, giving you a screen similar to Figure 2-6.

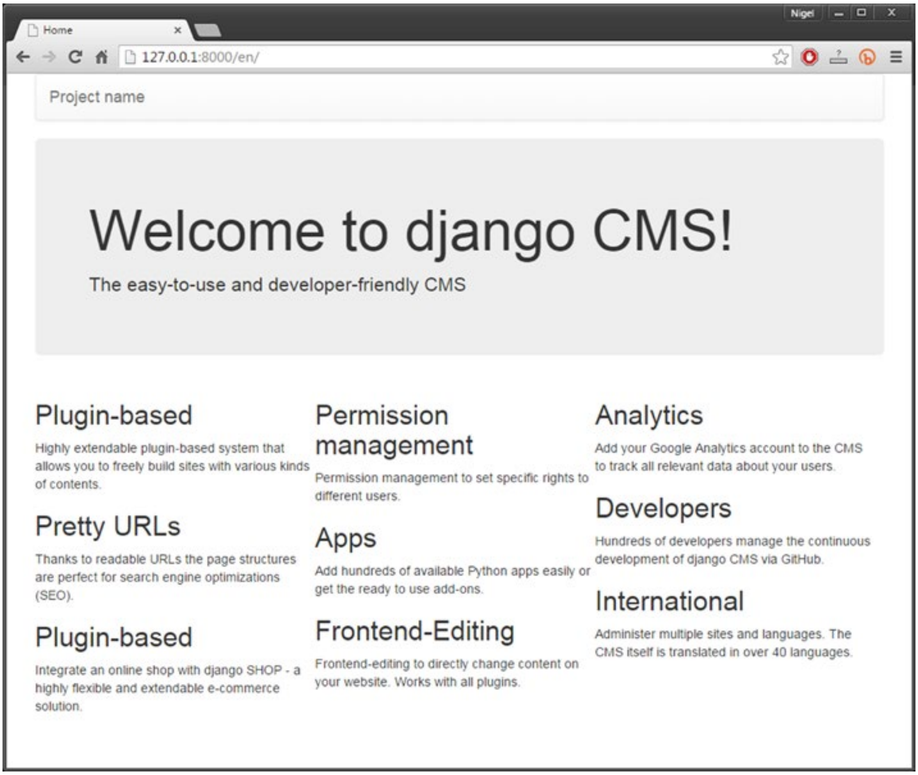


Figure 2-6. *Success!*

■ **Tip** Windows users might also want to consider installing Git BASH, which is part of the Git for Windows package (`https://git-for-windows.github.io/`). Most of the examples online assume you have some form of Unix running, so installing Git BASH will allow you to run Unix format command-line tools without needing to translate examples into Windows shell commands.

Summary

In this chapter, we have covered installing and configuring Python and django CMS, the result being a nice and clean (albeit empty) django CMS website. In the next chapter, we will explore the overall structure and design philosophy behind django CMS.

CHAPTER 3

Introduction to django CMS

There are a number of other CMS applications based on Django; however, at the time of writing, django CMS is the most popular[1] and is supported by a large community of users and developers.

By itself, django CMS is not a "plug and play" website builder where nonprogrammers can pick a template, add some content, and have a website up and running in a couple of hours. It has specifically been designed to give developers and designers a great deal more freedom to customize their websites.

Design Philosophy

Think of django CMS as being like a box of LEGO bricks compared to a model car kit—both can be used to build a model car, but with the LEGO you have almost total freedom to build any car you like, whereas with the kit, you will only get the car in the box (assuming you follow the instructions).

It's also helpful to understand that django CMS is content-agnostic in that it does not impose any set structure on your content. django CMS is designed to provide the publishing framework for your content, but how that content is structured is up to you; you have the freedom to do everything from formatting text on a page with plug-ins to dropping in entire Django applications and hooking them to django CMS menus and administration.

django CMS inherits much of its design philosophy from Django; it does all of the following:

- Implements a Model View Controller (MVC) architecture to separate presentation logic from business logic.
- Provides a pluggable architecture that allows easy addition of self-contained plug-ins and applications (apps) to expand the functionality of a website.
- Uses search engine friendly URLs by default.
- Supports multiple languages by default.

[1] https://www.djangopackages.com/grids/g/cms/

To enhance the content management capabilities of Django, it adds these features:

- Front end editing—allowing for structural and content modifications without needing a dedicated administration back end.
- Full touch interface support, with a focus on tablets and touch laptops from leading vendors. Many phones will also work, but you should note that they are explicitly not supported by django CMS's touch capabilities.
- A range of prebuilt plug-ins for common content management tasks (such as page editing) and third-party application integrations (such as social media).
- Versioning (history) for all content pages.
- Easy permission management for content creators—assignable create/edit/publish authorizations.
- Media asset management—files, images, videos, and animations (Flash).

Of course, no software package is perfect: django CMS does have disadvantages, some because of limitations in Django and some because of the way django CMS has been designed:

- It's not designed for simple content management out of the box; you can't just create a page and start entering content like some other CMS applications. You first need to have a template with a placeholder; then you have to add at least one plug-in, configure the plug-in, and add your content.
- It's overly complex for really simple sites as there is no easy way to detach a Django project from the database, so even if you don't need a database, you still need to configure one.
- There are not a large number of third-party plug-ins available at the time of writing. The number of available plug-ins is growing all the time as django CMS grows in popularity, but there are many functions that are readily available in more mature CMS applications that you will have to code yourself in django CMS.
- Because django CMS and Django are both open source projects that are in active development, backward-compatibility with some plug-ins and Django apps is problematic on occasion.

These disadvantages are very much offset by the power and flexibility of django CMS. Typically you would choose django CMS for applications where the benefit of building a fully customized CMS significantly faster than building one from scratch outweighs the need to be more active managing the more technical aspects of the CMS.

The Structure of django CMS

django CMS is structured to allow the majority of CMS management functions to be performed from the front end of the website, rather than have a separate administrative back end. django CMS is different from most other CMSes in that it allows you to edit content and structure from the front end, which makes updating, previewing, and publishing basic design and content changes very easy.

Accessing the site editor is as simple as appending the string `?edit` to the homepage URL—in our example, this will be `http://127.0.0.1:8000/en/?edit`. If you enter the URL correctly, you will see the front page of the demo website change as in Figure 3-1.

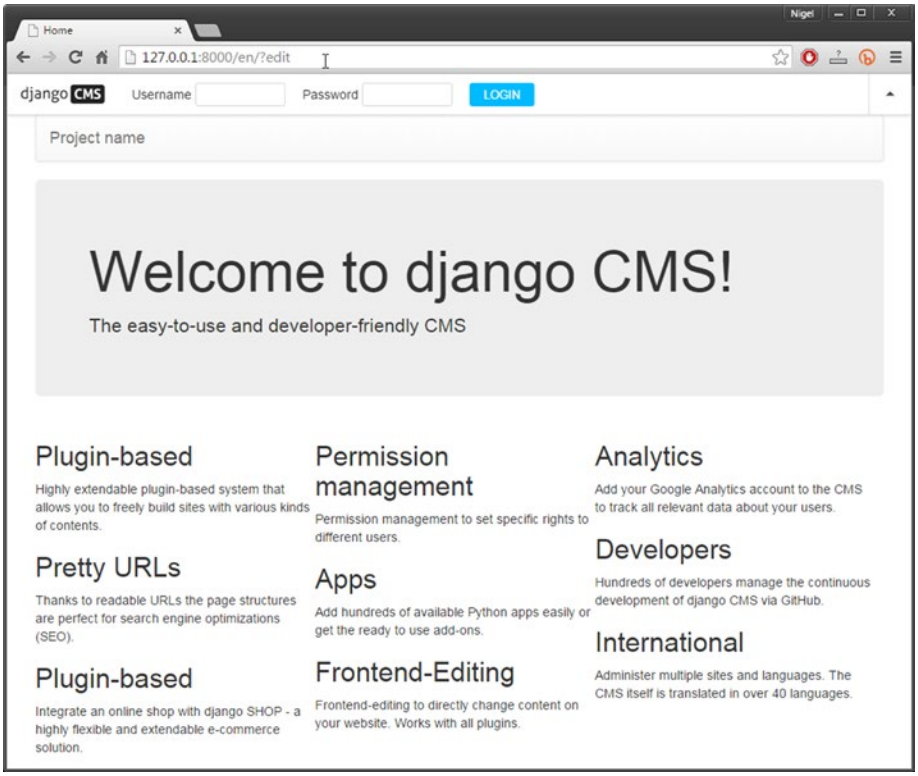


Figure 3-1. *DjangoCMS home page with edit toolbar displayed*

Log in to your demo website with the admin username and password you entered in Chapter 1. Once you are logged in, the toolbar will change to the logged-in state, as in Figure 3-2.



Figure 3-2. *django CMS toolbar changed to edit mode*

Once django CMS is in edit mode, you can access a range of functions in the toolbar via the dropdown menus. The available options for each menu will depend on your access level. At the moment, you are logged in as the administrator, so you will have full access to the site. We will explore the functions in the toolbar in more detail in the next section.

You are also able to edit text and page structure directly. Hovering your mouse over (or tapping) certain sections of your page will bring up a contextual edit link that allows you to edit both content (Figure 3-3) and structure (Figure 3-4).

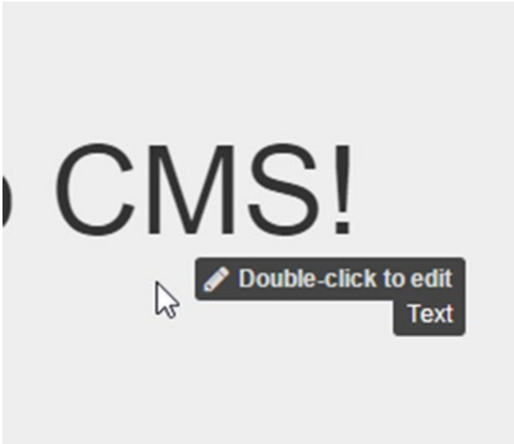


Figure 3-3. *Link to edit page content*

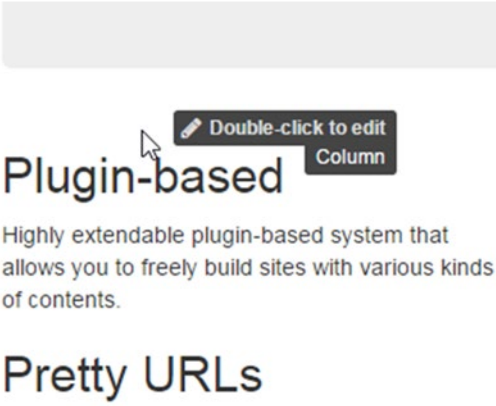


Figure 3-4. *Link to edit page structure*

You will see how to edit page structure and content in Chapter 8.

The django CMS Toolbar

For content and layout editors, and the site administrator, the django CMS toolbar provides all the tools necessary to manage and update a website via four dropdown menus:

- **The Site Dropdown** (example.com in the default install) has a number of menu options that allow the site administrator to manage pages, users, and the Django administration back end. It is not usually accessible to content and layout editors. We will explore the site dropdown in more detail in Chapter 8.
- **The Page Dropdown** provides access to all of the page functions, including adding pages, editing, publishing, and deleting pages. Depending on the permission settings, editors can also change the page template, page permissions, and a number of advanced settings. Chapter 8 also covers the Page dropdown.
- **The History Dropdown** provides access to the complete page history, with the ability to review page changes and to revert to an earlier version of the page. More on this in Chapter 8.
- **The Language Dropdown** allows you to manage various translations of a page. In this introductory book, we will use only English, so we will not be exploring the function of this menu.

The four buttons on the right of the toolbar are all related to page structure and editing and will be covered in detail in Chapter 8.

The django CMS Side Pane

There is one more structural element to the editing interface we need to cover before we move on to building our demo site: the side pane. If you select example.com ➤ Pages, you will see the side pane pop out as in Figure 3-5.

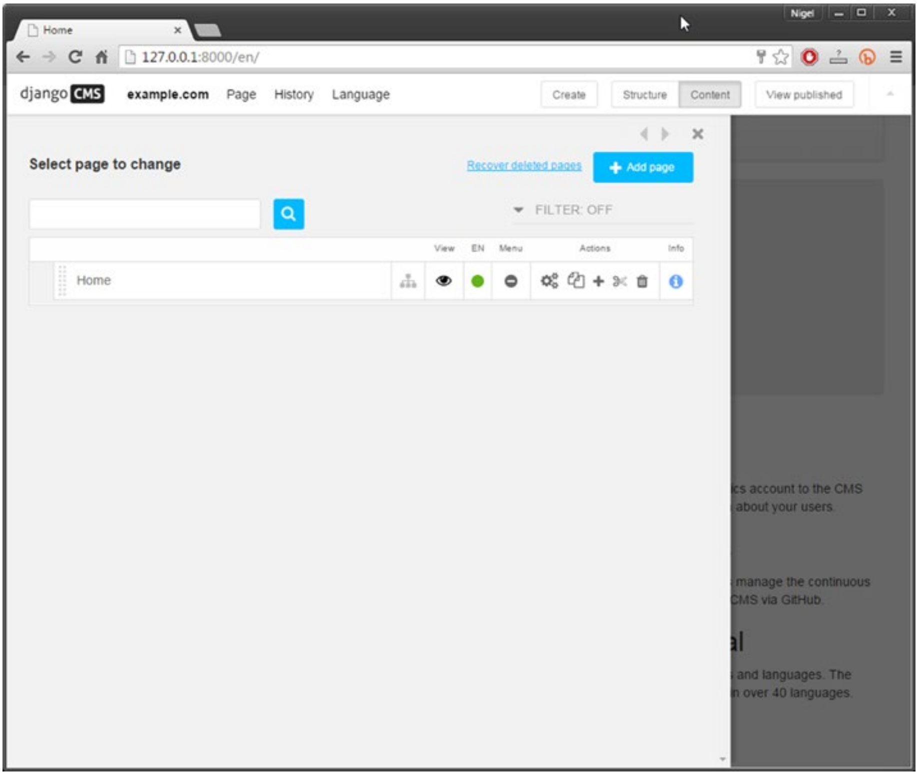


Figure 3-5. *The Side pane*

A few of the dropdown menu option will open the side pane to allow you to enter additional information. You will notice the three small navigation tabs on the top right of the side pane. Use the < > arrows to show and hide the side pane and the X to close it.

django CMS for Designers and Developers

Website designers and developers will have noticed that there is very little design and no coding at all available from the front end. This is by design and follows the same philosophy as Django—design, code, and content are three distinctly different disciplines and as such, need to be separated from each other.

But don't worry; django CMS is just as easy to work with for designers and developers as it is for content editors:

- **For designers**, django CMS uses a clean, simple HTML-based template system that does not require any coding. It also fully supports Bootstrap and rich content. Chapters 4 and 5 cover this in detail.
- **For developers**, django CMS follows Django's pluggable, application oriented architecture. django CMS achieves this through plug-ins and Django apphooks. We will explore these topics in detail in Chapters 6, 7, 9, and 10.

Summary

In this chapter, we have taken a high-level view of django CMS, looking at its structure, functions, and design philosophy. We have examined its strengths and weaknesses and explored how its design meets the separate needs of designers and developers.

In the next chapter, we will explore django CMS site templates in depth, introducing you to the django CMS implementation of Django's templating system, as well as its custom template tags, and we will lay the foundation for creating the site templates for your own blog website.

CHAPTER 4

Site Templates

In this chapter, we will explore django CMS site templates in depth. We will start with a detailed review of the technologies upon which django CMS templates are built. We will close the chapter by stepping through the templates created by the django CMS installer and tying them to what you have learned.

Don't worry if the theory scares you a little at first; in the next chapter we start the really fun stuff, but don't be too quick to jump ahead. Once you absorb all the material in this chapter, you will never complain about not understanding templates again!

Django Templates 101

django CMS inherits and extends Django's templating system, so it is important to cover the core principles behind the design of that system, namely:

- Inheritance
- Separation of logic and presentation
- Extensibility

Inheritance

Most modern websites have a common, site-wide design in which certain elements on the page—like the header, footer, and menus—are repeated on each page. The Django template system uses a design pattern in which each page inherits design properties from a single base template that is named, by convention, `base.html`. (You will find the `base.html` template created by the django CMS installer in `/mysite/templates` directory.)

As a quick refresher, let's look at how Django's template inheritance works. The base template is a complete HTML file; Listing 4-1 illustrates a minimal but valid implementation of a `base.html` file.

Listing 4-1. A Minimal `base.html` File

```
<!doctype html>
<html>
    <head>
        <title>{% block title %}Default Title{% endblock title %}</title>
    </head>

    <body>
        {% block content %}
          This is the content in the base template
        {% endblock content %}
    </body>
</html>
```

Figure 4-1 shows how this basic template will render in a browser.



Figure 4-1. *Rendering of a simple base template*

This minimal example introduces the key element behind Django's inheritance: the *block tag*. Anything between the `{%block ... %}` and `{%endblock ... %}` tags can be overridden in a child template that inherits from `base.html`. For example, let's consider a basic child template (Listing 4-2).

Listing 4-2. Basic Child Template

```
#child.html

{% extends "base.html" %}
{% block title %}Title of Child Page{% endblock title %}

{% block content %}
    <p>This text will override the block content tag in base.html </p>
{% endblock content %}
```

Figure 4-2 shows how the child template will render in a browser. Notice that both the content and the title of the page have been overridden in the child template.



Figure 4-2. *Child template overrides the block tags in base template*

The element that does all the magic here is the `{% extends "base.html" %}` tag. This tells the template renderer that `child.html` extends (inherits) the `base.html` template. When the renderer finds a block tag in the child template that is the same as the base template, the block in the child is sent to your browser.

Or to put it another way, when `child.html` is rendered in your browser, its title and content blocks will *override* the blocks in the base template, thus allowing each page to render custom content without having to duplicate or modify the site-wide content. Pretty cool, eh?

To see how inheritance works in a real live template, take a look at `page.html` and `feature.html` in `mysites/templates`. Don't worry if a lot of it doesn't make sense right now, because we will go through these templates in greater detail shortly.

Separation of Logic and Presentation

Django does not allow the execution of arbitrary code in page templates. This means that you cannot write code into the page, assign variables, or execute any kind of program logic in a template. There are some very good reasons for this, which I covered in Chapter 1; however, the most important with regard to design, and worth mentioning again here is that Django assumes (correctly!) that most web designers are not programmers and that it is extremely important not to mix program code and HTML.

Django does, however, provide a number of template tags and *template filters* that allow you to make presentation-related decisions. If you are relatively new to Django, the built in templating system is quite simple and consists of four main tag types:

- Variables have the format `{{ value }}`. When the template engine encounters a variable, it evaluates it and replaces it with a result. Variables can be any Python data type.
- Filters have the format `{{ value|filter }}`. Filters are used for modifying how variables are displayed.

- Tags have the format {% tag %}. They are very versatile and can be used by the template engine to
  - Create text in the output
  - Control program flow and perform logic
  - Load external information
- Comments have the format {# comment #}. Their function is similar to HTML comments: any text in the comment block will remain in the template to provide information to the template designer or programmer but will not be rendered to the browser. This format works only for single-line comments; for multiline you must use the {% comment %}...{% endcomment %} block tag.

Django's template tags provide a number of useful functions for presenting data from your application. For example, for/endfor tags allow easy iteration over list items, and if/else/endif tags allow simple decision make on what content is displayed. Django's template filters allow for sorting, filtering, and formatting of data. Filters are applied directly to template variables separated by the pipe character (above the \ on most keyboards); for example, {{ title|upper }}. You can also create libraries of your own custom template tags. Custom template tags are beyond the scope of this book; however if you are interested, more information can be found at https://docs.djangoproject.com/en/1.8/howto/custom-template-tags/. Table 4-1 summarizes Django's most commonly used template tags and filters .

Table 4-1. *Common Django Template Tags and Filters*

Common Django Template Tags	Description
{% if %} {% endif %}	Renders content between tags if statement is true. Example: {% if article_list %} There are some articles published {% endif %}
{% elif %} {% else %}	Optional tags for if/endif flow control tags. Provide same functionality as Python elif and else statements. Example: {% if article_list %} There are articles published {% elif article_drafts %} There are some articles in draft {% else %} No articles sorry! {% endif %}

(*continued*)

Table 4-1. (*continued*)

<table>
<tr><th>Common Django Template Tags</th><th>Description</th></tr>
<tr><td>{% for %}
{% endfor %}</td><td>Allows easy iteration over a list of items. Example:

  {% for article in article_list %}
    {{ article.title }}
  {% endfor %}
</td></tr>
<tr><td>{% empty %}</td><td>Optional tag to render an error or message if the list in a for/endfor statement is empty. Example:
... for statement
{% empty %}
<p>No articles sorry!</p>
{% endfor %}</td></tr>
<tr><td>upper
lower
title</td><td>Uppercase, lowercase, and title case formatting. Example:
{{ title|upper }}
If title is “Beginning Django,” the output will be “BEGINNING DJANGO.”</td></tr>
<tr><td>date</td><td>If the variable is a datetime object, it will format according to the format string. Example:
{{ today|date: "D d M Y" }}
would output Sat 19 Sep 2015.
The date format strings follow the same convention as PHP format strings. This was implemented to make it easier for designers to switch from PHP to Django.</td></tr>
<tr><td>Default</td><td>Outputs a default value if a variable is empty. Example:
{{ value|default:"empty" }}
If value is blank, the output will be the string “empty.”</td></tr>
<tr><td>first
last</td><td>Returns the first (or last) item in a list. Example:
{{ article_list|first }}
would output the first article from the list of articles.</td></tr>
<tr><td>truncatechars
truncatewords</td><td>Truncates the variable to the designated number of characters/words. Example:
{{ title|truncatechars:7 }}
would truncate “This is the Title” to “This is”, and
{{ title|truncatewords:3 }}
would output “This is the.”</td></tr>
</table>

This is only a small sample of all the Django template tags and filters available. Please see http://masteringdjango.com/django-built-in-template-tags-and-filters/ for a complete list.

Extensibility

Django's template system is designed to be extensible. It is not important to go into detail right now on how exactly Django achieves this, but it is important to understand how django CMS leverages Django's extensibility; namely, with several extended tag libraries, including these:

- cms_tags
- sekizai_tags
- menu_tags

CMS Tags

There are 19 custom template tags implemented by django CMS (listed in Table 4-2). To load the django CMS tags, we use Django's load template tag:

```
{% load cms_tags %}
```

This load tag must go at the top of your base.html file.

Table 4-2. *django CMS Custom Template Tags*

django CMS Custom Tags	
placeholder	render_plugin
static_placeholder	render_plugin_block
render_placeholder	render_model
render_uncached_placeholder	render_model_block
show_placeholder	render_model_icon
show_uncached_placeholder	render_model_add
page_lookup	render_model_add_block
page_url	page_language_url
page_attribute	language_chooser
	cms_toolbar

We will be using a number of these tags later in the book, but right now these are the three most important custom tags to examine:

- placeholder
- static_placeholder
- cms_toolbar

placeholder

The placeholder tag defines a placeholder within the template. A placeholder serves as a container where you can place plugins when editing a page; for example, this tag:

```
{% placeholder "content" %}
```

puts a placeholder within your template that will render as an empty container when you first add a page based on the template. You can then add a plugin to the container from the front-end editor to render content on the page.

The placeholder accepts any plugin that is available in the editor, and plugins can be nested. So, for example, you can add a Multi Columns plugin to add two columns and then fill each column with a Text plugin to render two columns of text on the page. You will see how to do this a bit later in the book.

The name of the placeholder does not matter. I have used the name "content" here because it's best practice to make your variable names as descriptive as possible, but you could have called the placeholder anything. On the subject of placeholder names, note that a placeholder is unique to the page, so you can have a placeholder named "content" on two different pages and they would refer to different objects. If you want a placeholder to be common to more than one page, you need to use static_placeholder.

You can also render additional content if the placeholder is empty using the or argument. To use r, you must add an {% endplaceholder %} closing tag. For example:

```
{% placeholder "content" or %}Coming Soon!{% endplaceholder %}
```

The placeholder tag can also inherit from any parent pages above it that have a placeholder with the same name. So a template containing this tag:

```
{% placeholder "content" inherit %}
```

would inherit the content placeholder from the first parent page above it in the inheritance tree that contains a placeholder named "content".

It is possible to combine inherit with or to create a complete fallback solution for inherited pages:

```
{% placeholder "content" inherit or %}No content anywhere!{% endplaceholder %}
```

static_placeholder

The `static_placeholder` tag is similar in function to the `placeholder` tag; however, it can be used anywhere in a template and is normally used to display content anywhere on your site, including inside other applications. For example, if we create this placeholder:

```
{% static_placeholder "footer" %}
```

and place it inside a template file (it does not matter which one), and add a text plugin and the text "`Copyright 2015`" from the front-end editor, every page that implements a static placeholder named "`footer`" will also display "`Copyright 2015`".

Of course, it makes more sense to put the static placeholder in your `base.html` file, especially if it is something like a footer or header, but there is no requirement that you do so. The other advantage is that if you decide to change the content of the placeholder, you can edit it from any page and it will update the rest automatically; you don't have to go back to the base page to edit it.

The `static_placeholder` tag also takes the `or` argument and an additional `site` argument to allow a static placeholder to be site-specific in a multisite installation. For example:

```
{% static_placeholder "footer" mysite or %}
    No footer on mysite.
{% endstatic_placeholder %}
```

Note that multisite installations are outside the scope of this book, but this example does give an idea of the easy scalability of django CMS.

cms_toolbar

The `cms_toolbar` custom tag enables the template engine to render the django CMS toolbar. The `cms_toolbar` tag should be placed just after the `<body>` tag in `base.html` and must not be enclosed in block tags:

```
# template header information

. . .

<body>
    {% cms_toolbar %}

# rest of the template

. . .
```

Sekizai Tags

Django-sekizai is a third-party library that uses blocks to render static media files like CSS and JavaScript (sekizai is the Japanese word for "blocks"). Sekizai allows designers to extend and/or add CSS and JavaScript files to templates without having to edit the underlying Python files.

Django-sekizai provides `css` and `js` namespaces that allow Django views to add media without altering templates; for example, the `cms_toolbar` custom template tag automatically adds CSS and JavaScript to the sekizai namespaces without any need to modify templates. Django-sekizai also automatically removes duplicates, so there is no risk of bloat when media files are included multiple times.

Because Django-sekizai is required by django CMS, the installer will have added the necessary modules and settings to your project. To include Django-sekizai in your template, you load it at the top of your `base.html` file:

```
{% load sekizai_tags %}
```

Django-sekizai is extremely simple to use; to add it to your templates, you only need two tags:

- `render_block`
- `addtoblock`

However, there are some rules you must follow for Django-sekizai to work:

- `render_block` tags cannot be in an included template; they must be in your `base.html` file.
- `render_block` tags cannot be placed inside Django block tags.
- If `addtoblock` is being used in an extended template (uses the `{% extends ... %}` tag), it must be enclosed within block tags.

■ **Caution** Pay close attention to these rules, as they are usually the source of "Invalid block tag" errors when using Django-sekizai. For more information, see https://django-sekizai.readthedocs.org/en/latest/#restrictions.

In practice, these rules are quite easy to follow. Place `{% render_block 'css' %}` in the header and `{% render_block 'js' %}` just before the closing body tag of your `base.html` file. For example, our basic template from Listing 4-1 would now look like Listing 4-3.

Listing 4-3. Minimal base.html file with Sekizai Tags Added

```
{% load sekizai_tags %}
<!doctype html>
<html>
    <head>
        <title>{% block title %}Default Title{% endblock title %}</title>
        {% render_block 'css' %}
    </head>

    <body>
        {% block content %}{% endblock content %}
        {% render_block 'js' %}
    </body>
</html>
```

Then, when you want to add CSS or JavaScript files to one of your child templates, you just need to make sure `addtoblock` is enclosed in block tags (Listing 4-4).

Listing 4-4. Basic Child Template With Sekizai Tags

```
#child.html (fragment)

{% extends "base.html" %}
. . .
{% block "content" %}
{% addtoblock "js" %}

# Add extra scripts here
. . .

{% endaddtoblock %}
{% endblock %}
```

Menu Tags

The `menus` application implements four additional custom template tags for rendering menus in django CMS:

- `show_menu`
- `show_menu_below_id`
- `show_sub_menu`
- `show_breadcrumb`

The tag we are most interested in right now is `show_menu` (we will be covering the rest in Chapter 10). To enable the menu tags in a template, we use Django's load tag:

```
{% load menu_tags %}
```

show_menu

The `show_menu` tag renders the default navigation menu for the page. The location of the menu is dependent on the design of the template. In the demo site created by the django CMS installer, the default menu is displayed horizontally along the top of the page, as shown in Figure 4-3.

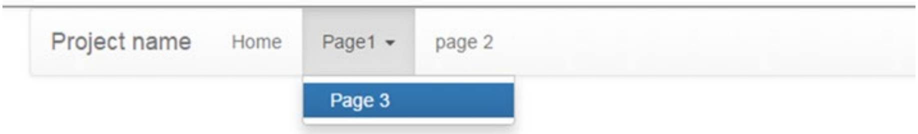


Figure 4-3. *The Default django CMS menu*

I have added a couple of pages, as well as a subpage in this example. Notice that the menu automatically formats the menu titles and creates dropdowns of subpages; the template engine creates the menu (with a bit of help from Bootstrap) without any input from you. We will dig into Bootstrap a bit more shortly. Adding pages will be covered a bit later in the book.

The `show_menu` tag takes eight optional arguments:

- **`from_level`**. The node level to start rendering the menu from. Default is `"0"` (root node).
- **`to_level`**. Node depth to render. Default is `"100"`, which in a practical sense means all child nodes.
- **`extra_inactive`**. The number of nodes to render under the current inactive node. The default is `"0"`, which effectively hides all of the child nodes under the current active menu item.
- **`extra_active`**. The number of nodes to render under the current active node. The default is `"100"`, which effectively expands all of the child nodes under the current active menu item.
- **`template`**. If a template is specified (for example `"menu.hml"`), django CMS will render the file `/templates/menu.html` in your application folder. If it is not specified, django CMS will render the default `menu.html` file (hidden inside Django's `site-packages` directory). See the accompanying caution for more on this.

- **namespace**. The namespace of your template. Template namespaces are beyond the scope of this book.
- **root_id**. ID of the menu root page. This is used when rendering child menus.
- **next_page**. This will retrieve all the children of the page ID specified and render them in the menu.

■ **Caution** It's best practice to always specify a template when using `show_menu`. One of the core philosophies of Django is portability, so all the templates specific to an application should reside within that application's template directory. If you don't do this, a change to the django CMS default template could break your site.

Because it is easier and neater to pass in positional arguments (rather than keyword arguments), the most common form of `show_menu` for a vertical menu is this:

```
{% show_menu 0 100 0 100 "menu.html" %}
```

which will show the whole menu from the root, with inactive submenus collapsed and the active menu expanded. For a horizontal menu (like our demo application), it looks like this:

```
{% show_menu 0 1 100 100 "menu.html" %}
```

which will display only the top-level pages in the menu and expand the children on the active menu item.

You might be asking yourself why `extra_inactive` is `100` and not `0`, because setting it to `100` would expand all children of the inactive menu items, which we don't want—right?

Yes, but we are using this trick to take advantage of one of the features of Bootstrap—inactive nodes are collapsed automatically, and Bootstrap displays the dropdown arrow icon so you know that the parent menu item has children. If you set `extra_inactive` to `0`, you will not see the dropdown arrow unless the menu item is active.

Bootstrap

Back in Chapter 1, I said that a modern website must be mobile-friendly to cater to the multitude of devices now connected to the Internet. But what exactly does *mobile-friendly* mean?

While there are many and sometimes contradictory answers to this question, flexibility is key: the number of types of mobile devices, as well as the operating systems they run, are incredibly diverse, and a website that will support as many devices as possible needs to be adaptable. To be flexible and adaptable, site functionality boils down to three simple requirements:

- **Responsiveness**. This is an industry term, but it means that your website will automatically resize and rearrange content depending on the device. A well-designed responsive website will provide the same positive user experience to a visitor using a smartphone as to a visitor browsing from a desktop with a 28" monitor.
- **Compliance**. Android, iOS, OSX, Windows, Linux: they all have different ways of doing things, incompatible system software, different fonts and multiple browser versions. A mobile-friendly site must be able to deliver content that renders correctly across disparate systems.
- **Speed**. Mobile data speeds are generally slower, and mobile devices are often light on processing power compared to desktops and laptops. If your site does not load fast, most mobile users will move on to a competitor's site.

While there is still a great deal of variety on the server side of the Internet, these requirements have pared the browser side down to three basic technologies—CSS, HTML, and JavaScript.

But using these technologies together to build a website from scratch is prodigiously difficult and has led to the rise of a number of front-end web frameworks that make the design, development and management of a website that uses these technologies much simpler.

Bootstrap, originally the brainchild of two Twitter employees, was released as an open source technology in 2011. In the four years since, it has become the 800-pound gorilla of front-end frameworks. Bootstrap is so popular that it is now the default framework for just about all CMSes and large custom sites implementing responsive design.

A full explanation of Bootstrap is well beyond the scope of this book; however, a brief introduction to some key concepts will make designing django CMS responsive templates much easier.

The Bootstrap Grid System

The Bootstrap grid is a 12-column, infinite-row fluid grid system that automatically reformats and scales itself depending on the device. The exact layout of the grid and how it behaves on each screen size is governed by predefined CSS classes (see the next section). Each cell of the grid contains an HTML `<div>...</div>` block in which you can place anything that you would legally be able to place between `div` tags in a

nonresponsive template. This is easier to illustrate using a simple example. Figure 4-4 shows how a simple page with two-columns and a right-menu might look on a desktop computer.



Figure 4-4. *Responsive template on a desktop computer*

On a smaller screen like a tablet, the design might look like Figure 4-5.

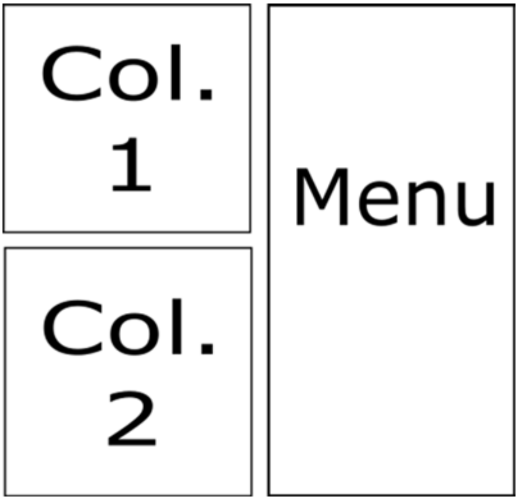


Figure 4-5. *Responsive template on a tablet*

And on a smartphone, the design might look like Figure 4-6.

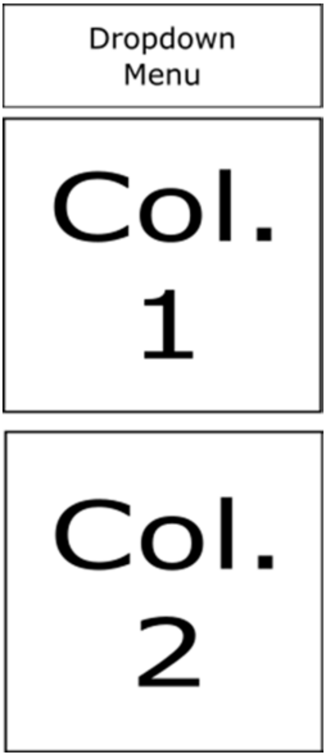


Figure 4-6. *Responsive template on a smartphone*

This is a really simple example, but Bootstrap is flexible and powerful enough to create extremely sophisticated designs, all without any program code. And how does it do that, I hear you ask? Well, that's what we will look at next—*CSS classes* and *components*.

Bootstrap CSS Classes

Bootstrap achieves its formatting magic with very sophisticated style sheets that define over 600 classes for formatting and styling HTML elements. These classes are grouped into eight functional groups:

- **Grid system**. Classes for formatting the grid depending on the device size—extra small (`xs`), small (`sm`), medium (`md`), and large (`lg`)—with column spanning modifiers. For example, the `col-xs-4` class spans four columns on an extra small screen (smartphone).

- **Typography**. Text styling, such as headings, body copy, inline text formatting, alignment, and special formatting like block quotes, addresses, and text transformations.
- **Code**. Inline and code blocks, user input, and variable formatting.
- **Tables**. Formatting and coloring, sizing, and mouse interactions.
- **Forms**. Sophisticated formatting for forms and form controls.
- **Buttons**. Display options, coloring, and sizing.
- **Images**. Image resizing and shaping.
- **Helper functions**. Miscellaneous classes for contextual coloring, clearfix, screen reader functions, and showing and hiding columns depending on screen size.

Bootstrap's CSS uses Less, a CSS preprocessor that adds a number of functions including variables, mixins, and functions for compiling CSS. Examples include these:

- Grayscale colors, such as, `@gray, @gray-dark, @gray-light`.
- Semantic colors, such as `@brand-primary, @brand_info`.
- Use of color variables in CSS, such as `background-color: @brand-primary`.
- Typographic bases, such as `@font-family-base`, `@font-size-base`.
- Typographic mixins, such as `@font-size-h1: floor((@font-size-base* 2.6));`
- Link styles, such as `@link-color, @link-hover-color`

We will not be exploring Bootstrap classes any further here; I just wanted to give you a taste of all the useful design elements you have at your disposal when using Bootstrap and help you understand that when you are building a site design with django CMS, you don't have to reinvent the wheel. To explore Bootstrap fully, please refer to `http://getbootstrap.com/css/`.

django CMS leverages the power of Bootstrap right out of the box, so not only do django CMS default templates take advantage of Bootstrap classes, but you can also implement them easily in your own templates.

Bootstrap Components

To bring our section on Bootstrap to a close, we just need to touch on one more thing briefly—components. Bootstrap components are reusable bits of code that are generally a mix of CSS and JavaScript. django CMS uses a number of Bootstrap components, including dropdowns, navs, alerts, and the jumbotron. Other components include these:

- Glyphicons
- Button groups and dropdowns

- Pagination
- Breadcrumbs
- Thumbnails
- Progress bars
- Panels and wells

■ **Note** We have barely scratched the surface with Bootstrap. While it is not necessary to dig any deeper into Bootstrap to understand the rest of this book, your future as a professional programmer would be greatly benefitted by bookmarking `getbootstrap.com/getting-started/` and spending a few hours gaining an in-depth understanding of Bootstrap and its capabilities.

Tying it all Together

We have covered a lot of ground in this chapter, and now it is time to tie together all the things you have learned. We will do this by examining the template files that were generated by the django CMS installer.

Assuming you used the same file names as the installation examples in Chapter 2, you will find these templates in

```
C:\users\<yourusername>\myCMS-tutorial\mysite\templates\
```

We will start with the `base.html` file, which is shown in Listing 4-5 (Note: some of the URLs have been shortened to improve readability):

Listing 4-5. Complete sample `base.html` file

```
{% load cms_tags staticfiles sekizai_tags menu_tags %}
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>
      {% block title %}
        This is my new project home page
      {% endblock title %}
    </title>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link rel="stylesheet" href="https://cdnjs ... bootstrap.min.css">
    <link rel="stylesheet" href="https:// ... bootstrap-theme.min.css">
    {% render_block "css" %}
```

```
15.    </head>
16.    <body>
17.      {% cms_toolbar %}
18.      <div class="container">
19.        <div class="navbar navbar-default" role="navigation">
20.          <div class="navbar-header">
21.            <button type="button" class="navbar-toggle" ➥
         data-toggle="collapse" data-target=".navbar-collapse">
22.              <span class="sr-only">Toggle navigation</span>
23.                   <span class="icon-bar"></span>
24.                   <span class="icon-bar"></span>
25.                   <span class="icon-bar"></span>
26.                 </button>
27.                 <a class="navbar-brand" href="#">Project name</a>
28.              </div>
29.              <div class="navbar-collapse collapse">
30.                  <ul class="nav navbar-nav">
31.                      {% show_menu 0 1 0 100 "menu.html" %}
32.                  </ul>
33.              </div>
34.          </div>
35.          {% block content %}{% endblock content %}
36.      </div>
37.      <script src="https://cdnjs ... jquery.min.js"></script>
38.      <script src="https://cdnjs ... bootstrap.min.js"></script>
39.      {% render_block "js" %}
40.    </body>
41.  </html>
```

Now that you have the complete listing in front of you, let's step through the most important parts:

- **Line 1**. Django's load tag accepts multiple arguments, so we are loading all of the extended tag libraries in one go.
- **Lines 7 to 9**. Here Django's block tags are used to define the title block for the page. This code will be overridden by any child template that defines a title block.
- **Line 14**. This is the Sekizai render_block tag to render the CSS at the top of the file.

- **Line 17**. This is where we load the django CMS toolbar into the page.
- **Lines 19 to 34**. This is the page navigation bar implemented in Bootstrap. For a navigation bar to show anything, you must pass it a list, which we do on Line 31 with our `show_menu` tag. For the curious, lines 21 to 26 are how Bootstrap renders the "hamburger" icon that shows up instead of the full menu on small screens.
- **Line 35**. Here Django's block tags are used to define the content block for the page. This will be overridden by any child template that defines a content block.
- **Line 39**. This is the Sekizai `render_block` tag to render the JavaScript at the bottom of the page. Using it is best practice to increase page load speed.

As you can see, this is all very easy to follow once you know what each line is doing. As an exercise, I have left the three remaining sample templates—`page.html, feature.html` and `menu.html`—for you to examine on your own to improve your understanding, but here are some quick notes to help you along:

- `Page.html` and `feature.html` correspond to the `page` and `page with feature` templates in the Django editor front-end. We will dig deeper into page templates in the next chapter.
- Both `page.html` and `feature.html` extend `base.html`.
- You will notice that our sample home page (Figure 2-6) uses the `feature.html` template. The banner at the top is rendered by Bootstrap's `jumbotron` class. Add a new page and select the "page" template, and you will see a blank page template (Figure 4-7).
- Note the use of django CMS placeholder tags in `page.html` and `feature.html`.
- Notice how `menu.html` uses Django's `if/endif` and `for/endfor` tags to iterate over all the pages listed in the menu.

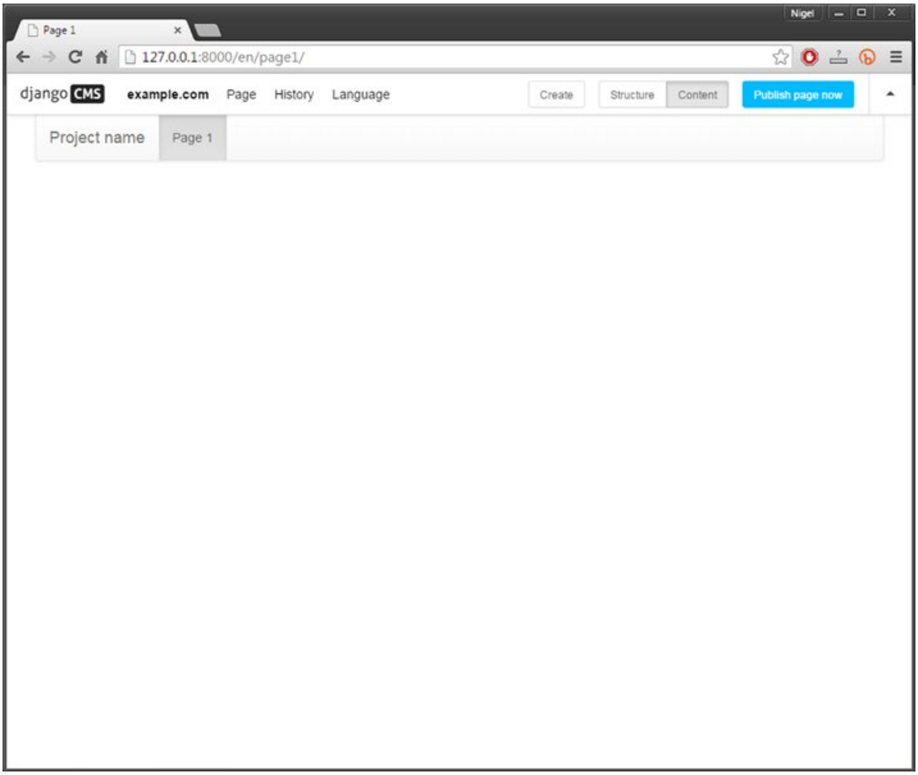


Figure 4-7. *The blank page template*

Summary

That's it for this chapter. We have covered the templates in great detail, and you should now have a very solid grounding in how django CMS templates are built. You should also be comfortable with the basics of Bootstrap and how django CMS leverages Bootstrap's CSS classes to create a modern site template that will look good on the majority of devices.

In the next chapter we are moving on to the fun stuff—designing your own templates.

CHAPTER 5

Your Blog Website: Templates

Now that you have a thorough grounding in how the django CMS templates are constructed, it is time to build the templates for your blog website. In this chapter, you will:

1. Start with a fresh, empty install of django CMS.
2. Create your own `base.html` template.
3. Create a CSS file to apply custom styling to your blog.
4. Create a new template, `blogpage.html`, that will display a single blog post along with a right sidebar menu listing all posts.
5. Edit your setting file to apply the new templates to your blog site.

Note All of the source code in this chapter is available for download from the book's website; however, it is recommended that you type out the files yourself the first time through. It can be tedious sometimes, but you will definitely learn more by doing so.

Create a Clean Install

We want to start with a completely empty django CMS installation for our blog site, so the first step is to create a new virtual environment (call it `myBlog`) and install django CMS in a new project directory called `myBlogProject`.

The steps to creating the new installation are almost identical to those you took in Chapter 2, but I will step through them briefly to refresh your memory (all commands are entered at the command prompt):

1. Enter `Virtualenv myBlog`.
2. Enter `myBlog\scripts\activate`, and the command prompt should change to `(myBlog) C:\Users\YourUsername>`.
3. Enter `pip install djangocms-installer`.
4. Enter `mkdir myBlogProject`.
5. Enter `cd myBlogProject`.
6. Enter `djangocms -p . myblog`.
7. Configuration is almost the same: accept all defaults, except enter `en` for language and `yes` to use Bootstrap. This time we don't was the demo starting page, so we accept the default `no` for the last question.

I have included an edited version of the output from your command window in Listing 5-1 to give you an idea of what you should see. It won't be exactly the same, but it should be close. If all goes according to plan, when you navigate to `http://127.0.0.1:8000/` in your browser, you should be presented with the django CMS login page (Figure 5-1).

Listing 5-1. Command Window Output

```
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\Nigel>virtualenv myBlog
Using base prefix 'c:\\python34'
New python executable in myBlog\Scripts\python.exe
Installing setuptools, pip, wheel...done.

C:\Users\Nigel>myBlog\scripts\activate
(myBlog) C:\Users\Nigel>pip install djangocms-installer
Collecting djangocms-installer
  Downloading djangocms_installer ... #elipsis = additional text removed
...
Installing collected packages: ...
Successfully installed ...

(myBlog) C:\Users\Nigel>mkdir myBlogProject

(myBlog) C:\Users\Nigel>cd myBlogProject
```

```
(myBlog) C:\Users\Nigel\myBlogProject>djangocms -p . myblog
Database configuration (in URL format) [default sqlite://localhost/project.
db]:
django CMS version (choices: 2.4, 3.0, 3.1, stable, develop) [default
stable]:
Django version (choices: 1.4, 1.5, 1.6, 1.7, 1.8, stable) [default stable]:
Activate Django I18N / L10N setting (choices: yes, no) [default yes]:
Install and configure reversion support (choices: yes, no) [default yes]:
Languages to enable. Option can be provided multiple times, or as a comma
separated list. Only language codes supported by Django can be used here: en
Optional default time zone [default Australia/Sydney]:
Activate Django timezone support (choices: yes, no) [default yes]:
Activate CMS permission management (choices: yes, no) [default yes]:
Use Twitter Bootstrap Theme (choices: yes, no) [default no]: yes
Use custom template set [default no]:
Load a starting page with examples after installation (english language
only). Choose "no" if you use a custom template set. (choices: yes, no)
[default no]: no
Creating the project
Please wait while I install dependencies
  ...
Dependencies installed
Creating the projectOperations to perform:
  ...
Synchronizing apps without migrations:
  Creating tables...
  Installing custom SQL...
  Installing indexes...
Running migrations:
  ...
Creating admin user
Username (leave blank to use 'nigel'):
Email address: <your email>
Password: <enter a password>
Password (again):
Superuser created successfully.
All done!
Get into "C:\Users\Nigel\myBlogProject" directory and type "python manage.py
runserver" to start your project

(myBlog) C:\Users\Nigel\myBlogProject>
```

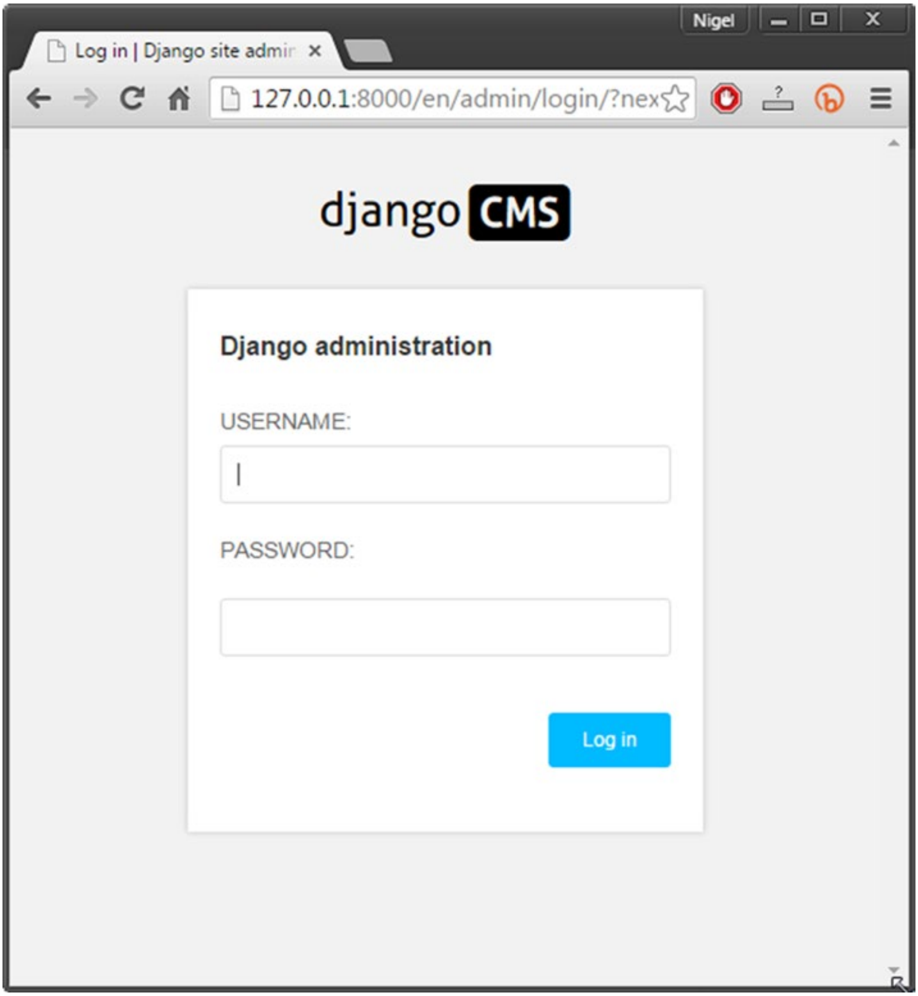


Figure 5-1. *A clean install of django CMS*

Once you have logged in, you will be greeted with the django CMS content creation wizard, a simple automated tool for adding content to your django CMS website. At the moment, we only have the default New Page content wizard (Figure 5-2). We will discuss content wizards in greater depth in Chapter 10.

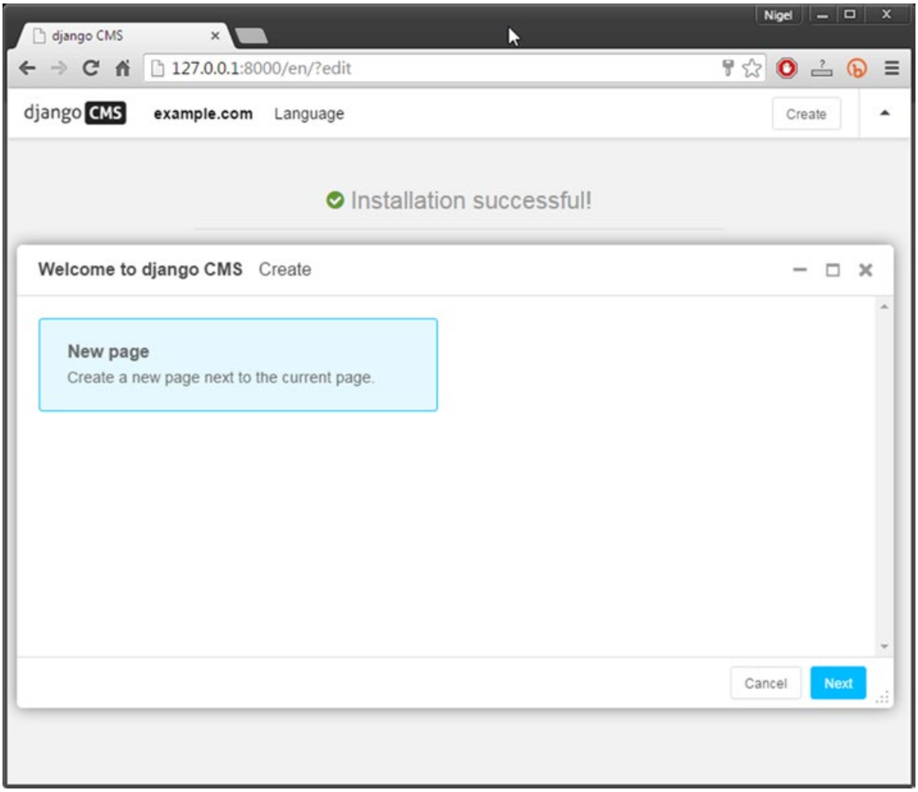


Figure 5-2. *The django CMS content creation wizard*

Go ahead and select Next to create your new page. Enter **Home** for the page title and some content if you want to. Content is optional at this stage, as we will be adding much more content to this page later on in the book. Select Create at the bottom right of your screen, refresh your browser, and your site should look like Figure 5-3.

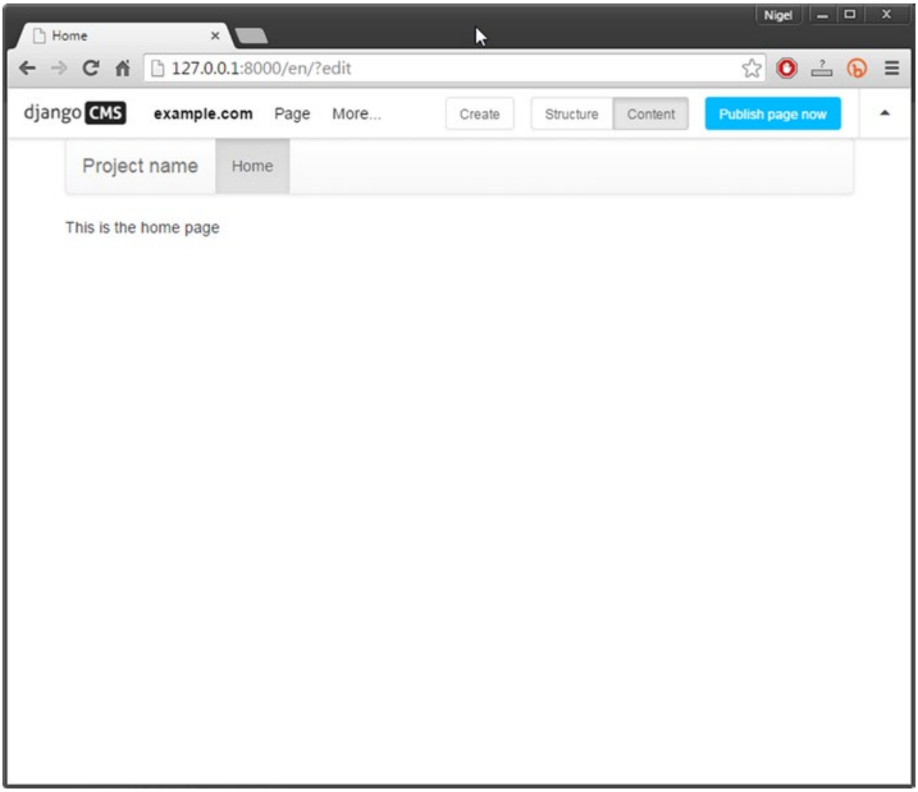


Figure 5-3. *Your new but completely empty website*

Create Your Base Template

Your blog site is looking rather empty and boring at this stage, so it is time to build our new templates. We are going to start with our base template. To make this happen, we need to

1. Rename `\myblog\templates\base.html` to `base.html.old`.
2. Create our new `base.html` file (Listing 5-2) and save it to `\myblog\templates\`.
3. Create our custom CSS file, `myblog.css` (Listing 5-3), and save it to `\myblog\static\`.

4. Create a 50×50 pixel logo and save it to `\myblog\static\logo.png`. It does not matter what the logo is, but a simple design on a transparent background works best. You can use a program like Microsoft Paint or GIMP to do this (or Photoshop if you have it). I have also included a simple logo in the source code if you don't want to create your own.

5. Download `ie10-viewport-bug-workaround.js`[1] and save it to `\myblog\static\`. This is a workaround put out by Bootstrap (introduced in Chapter 4) to solve a bug in Internet Explorer on Windows 10. If you are not planning to deploy your website, you don't have to add this file.

Listing 5-2. Our `base.html` File

```
{% load cms_tags staticfiles sekizai_tags menu_tags %}
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <link rel="icon" href="favicon.ico">

    <title>{% block title %}MyBlog Title{% endblock title %}</title>

    <link href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
    bootstrap/3.3.4/css/bootstrap.min.css" rel="stylesheet">

    <!-- MyBlog custom styles -->
    <link href="{% static "myblog.css" %}" rel="stylesheet">

    <!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and
    media queries -->
    <!--[if lt IE 9]>
      <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.
      js"></script>
      <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
    <![endif]-->
    {% render_block "css" %}
  </head>
```

[1] https://github.com/twbs/bootstrap/blob/master/docs/assets/js/ie10-viewport-bug-workaround.js

```
  <body>
    {% cms_toolbar %}
    <nav class="navbar navbar-default navbar-static-top">
      <div class="container">
        <div class="navbar-header">
          <button type="button" class="navbar-toggle collapsed" data-
          toggle="collapse" data-target="#navbar" aria-expanded="false"
          aria-controls="navbar">
            <span class="sr-only">Toggle navigation</span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
          </button>
          <a href="#"><img src="{% static "logo.png" %}" /></a>
        </div>
        <div id="navbar" class="collapse navbar-collapse">
          <ul class="nav navbar-nav">
          {% show_menu 0 1 0 100 "menu.html" %}
          </ul>
        </div>
      </div>
    </nav>

    <div class="container">
            <div class="blog-header">
                  <h1 class="blog-title">Not Just Another Blog</h1>
                  <p class="lead blog-description">My awesome blog built
                  with django CMS.</p>
            </div>
            {% block content %}{% endblock content %}
    </div>

    <footer class="footer">
      <div class="container">
        <p class="text-muted">{% block footer %}My Blog&copy;2015{% endblock
        footer %}</p>
      </div>
    </footer>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/1.11.3/
    jquery.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-
    bootstrap/3.3.4/js/bootstrap.min.js"></script>
    <!-- IE10 viewport hack for Surface/desktop Windows 8 bug -->
    <script src="{% static "ie10-viewport-bug-workaround.js" %}"></script>

    {% render_block "js" %}
  </body>
</html>
```

The code in Listing 5-2 is identical in function to Listing 4-5 from the last chapter, so please refer to that discussion for a more detailed line-by-line analysis. Note the following changes here:

- I have updated the HTML header to the current Bootstrap header. Basically, this adds some additional support for cross-browser compatibility and a different Content Delivery Network (CDN) for the Bootstrap scripts. You can get the latest Bootstrap templates (which you can use as the basis for your own django CMS templates) from http://getbootstrap.com/getting-started/#template.
- I have changed the navbar (top menu) class from `navbar-default` to `navbar-static-top` to prevent the django CMS toolbar from floating over the top of the navbar and obscuring your top menu.
- I have removed the "project name" text anchor and replaced it with a logo image.
- I have added a blog header before the page content. This gives your site a banner-type header that is common on many blogs.

Listing 5-3. The `myblog.css` File

```
html {
  position: relative;
  min-height: 100%;
}
body {
  /* Margin bottom by footer height */
  margin-bottom: 60px;
  font-family: Georgia, "Times New Roman", Times, serif;
  color: #555;
}
h1, .h1, h2, .h2, h3, .h3, h4, .h4, h5, .h5,h6, .h6 {
  margin-top: 0;
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
  font-weight: normal;
  color: #333;
}
.footer {
  position: absolute;
  bottom: 0;
  width: 100%;
  height: 60px;
  background-color: #f5f5f5;
}
.blog-header {
  padding-top: 20px;
  padding-bottom: 20px;
}
```

```
.blog-title {
  margin-top: 30px;
  margin-bottom: 0;
  font-size: 60px;
  font-weight: normal;
}
.blog-description {
  font-size: 20px;
  color: #999;
}
.sidebar-module {
  padding: 15px;
  margin: 0 -15px 15px;
}
.sidebar-module-inset {
  padding: 15px;
  background-color: #f5f5f5;
  border-radius: 4px;
}
.sidebar-module-inset p:last-child,
.sidebar-module-inset ul:last-child,
.sidebar-module-inset ol:last-child {
  margin-bottom: 0;
}
.pager {
  margin-bottom: 60px;
  text-align: left;
}
.pager > li > a {
  width: 140px;
  padding: 10px 20px;
  text-align: center;
  border-radius: 30px;
}
.blog-post {
  margin-bottom: 60px;
}
.blog-post-title {
  margin-bottom: 5px;
  font-size: 40px;
}
.blog-post-meta {
  margin-bottom: 20px;
  color: #999;
}
body > .container {
  padding: 15px 15px 0;
```

```
}
.container .text-muted {
  margin: 20px 0;
}
.footer > .container {
  padding-right: 15px;
  padding-left: 15px;
}
. row {
  margin-right: 0px!important;
}
```

Listing 5-3, `myblog.css`, is a standard CSS file. Going into detail on CSS is beyond the scope of this book. If you need to brush up on your CSS knowledge, a great resource can be found at `http://www.w3schools.com/css/`.

If all goes according to plan, when you reload your homepage, you will be greeted with a site that looks like Figure 5-4 (note that I have collapsed the django CMS toolbar in this screenshot so you can see the full template).

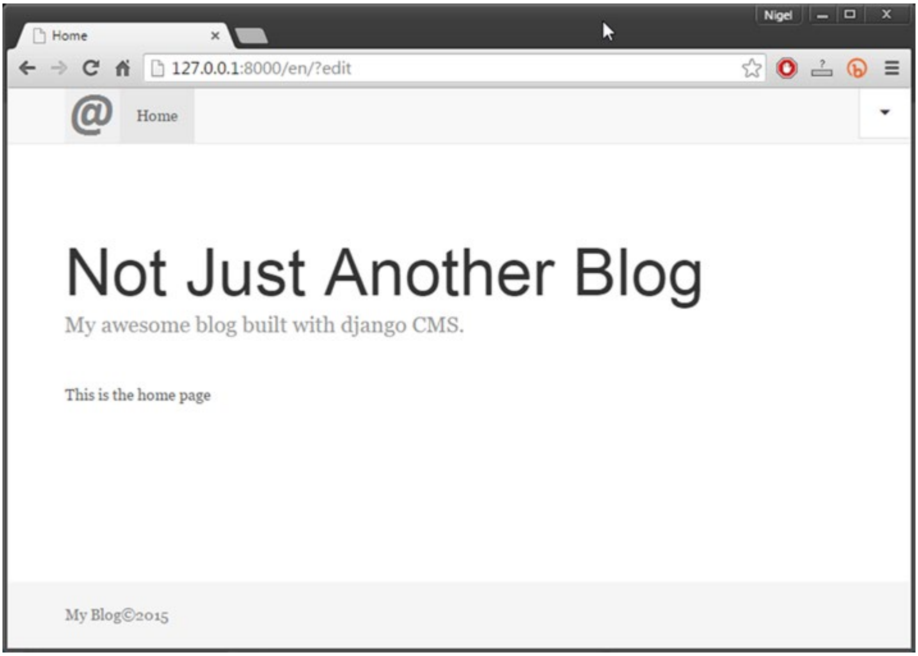


Figure 5-4. *Your New Blog Base Template*

Create Your Blog Templates

While our blog is looking much prettier than when we started, it still doesn't look much like a blog. For a start, there isn't the usual sidebar listing recent posts. In fact, there aren't even any posts! So that's what we are going to do now: Listing 5-4 shows our base template for blog pages, `page.html`. This page can be used as a basis for many different types of blog pages, not just blog posts (for example, a blog roll listing latest posts).

Listing 5-5 shows our blog post page, `blogpage.html`, which inherits from our `page.html` template.

You will notice that the sidebar is in these pages, not the base template. If you were wondering why, it's so we have maximum flexibility with our site. We want to be able to show pages without the sidebar (to create, for example, a full-screen reading page), or maybe even to show a completely different sidebar on certain pages. In the case of our blog, this makes turning it into a reusable application or django CMS plugin at a later date much easier.

Keeping your base template as generic as possible is almost always a good thing. Couple this philosophy with Django's powerful template inheritance capabilities, and you will find over time that you are collecting a portfolio of basic templates that allow you to meet even the most demanding design challenges without having to start from scratch every time.

Listing 5-4. Our `page.html` Template

```
{% extends "base.html" %}
{% load cms_tags sekizai_tags %}

{% block title %}{% page_attribute "page_title" %}{% endblock title %}

{% block content %}
<div class="row">

      <div class="col-sm-8 blog-main">
            {% block page-inner %}
              {% placeholder "content" %}
                {% endblock page-inner %}
      </div>

      <div class="col-sm-3 col-sm-offset-1 blog-sidebar">
    <div class="sidebar-module sidebar-module-inset">
    <h4>About</h4>
    <p>My name is Big Nige and I built this blog all by myself.</p>
    </div>
    <div class="sidebar-module">
    <h4>Latest Posts</h4>
     <ol class="list-unstyled">
      <li><a href="#">Blogpost 1</a></li>
      <li><a href="#">Blogpost 2</a></li>
     </ol>
```

```
      </div>
    <div class="sidebar-module">
    <h4>User</h4>
    <ol class="list-unstyled">
      <li><a href="?edit">Log In</a></li>
    </ol>
    </div>
  </div>
</div>
{% endblock content %}
```

The code in Listing 5-4 should be very easy to follow; a couple of things to note are:

- I am using Bootstrap's 12-column layout to create an 8-column content area with a single-column offset and then a 3-column right sidebar.
- I have provided some filler text to create a dummy menu in the sidebar. This is so we can check the template visually. It will be replaced by a placeholder in Chapter 9.
- I have added a Log In link at the bottom of the right sidebar to make it easier to log in, rather than having to add `?edit` to the URL every time you want to log in.

Listing 5-5. The `blogpage.html` Template

```
{% extends "page.html" %}
{% load cms_tags %}
{% load sekizai_tags %}

{% block title %}{% page_attribute "page_title" %}{% endblock title %}

{% block page-inner %}
<div class="blog-post">
  <h2 class="blog-post-title">{% page_attribute "page_title" %}</h2>
<p class="blog-post-meta">Published {{ ➥
request.current_page.creation_date|date:'d M Y' }} by ➥
<a href="#">{{ request.current_page.created_by }}</a></p>

    {% placeholder "content" %}
    {% static_placeholder "social" %}
</div>
{% endblock page-inner %}
```

Thanks to Django's template inheritance, our blog post template is very simple, as all the heavy lifting has been done by `base.html` and `page.html`. All `blogpage.html` does is replace the `{% page-inner %}` block tag with a formatted blog post that shows the post title and author and date information as a header and provides a placeholder for social

media share buttons that we will add later, in Chapter 9. Note that we are accessing some of the page attributes that are conveniently passed to the page context by django CMS's Title and Page models.

Now that we finally have all our templates written, we just need to register them with django CMS. Fortunately, this is very easy to do; just open up `myblog\setting.py` in your text editor and change the `CMS_TEMPLATES` setting to the following:

```
CMS_TEMPLATES = (
    ## Customize this
    ('page.html', 'Page'),
    ('blogpage.html', 'Blog Page with Sidebar'),

)
```

Save the file and refresh your homepage. From the django CMS toolbar, select Page ➤ Templates ➤ Blog Page with Sidebar, and your template should change to look just like Figure 5-5.

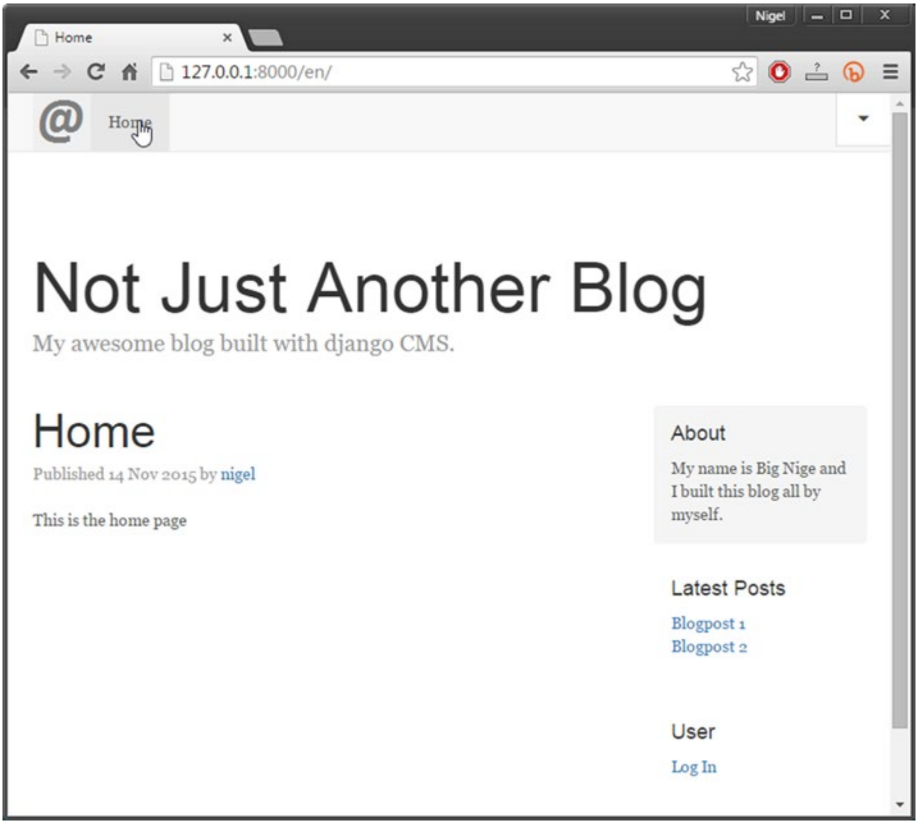


Figure 5-5. *Your completed blog template*

And that's it! Have a stretch and give yourself a pat on the back. That was an awful lot of typing (assuming you didn't cheat and download the files), but I hope you agree that the result was worth it.

Summary

In this chapter we have created a set of templates for your blog website; in doing so you have learned a bit more about Django's template inheritance and how Bootstrap makes it easy to create responsive page templates.

In the next chapter we will move on to actually creating some pages for your website, and I will introduce you to one of the most powerful features of django CMS: plugins.

CHAPTER 6

django CMS Plugins

Core to the flexibility of django CMS is an open structure that has been designed to be pluggable. Conceptually, this is different from most other CMS software available. With the majority of other CMSes, you can only control what is displayed on a page as a single entity (either a page or an article), and content is usually edited from an administrative back end.

django CMS takes a different approach—its plugin architecture provides flexible, drop-in containers that can be edited directly from the front end. These containers are designed for easy inclusion of all manner of content, from simple HTML tags to sophisticated embedded applications.

django CMS does not restrict the page layout; for example, you can drop a banner into the top of a page, write an article to go underneath it, and then drop a social media plug-in below the article. You are free to copy this layout to other pages, or to do something completely different on each page. We will get to the steps for doing this in Chapter 8. In the next two chapters, however, we will first explore all the plug-ins you already have at your disposal. In this chapter, you will be learning how the built-in django CMS plugins work.

Getting Started

The best way to understand how this works is to dive right in. Click on the Home page you created in the last chapter and then click the Structure button right of center in the django CMS toolbar. Your screen should look something like Figure 6-1.

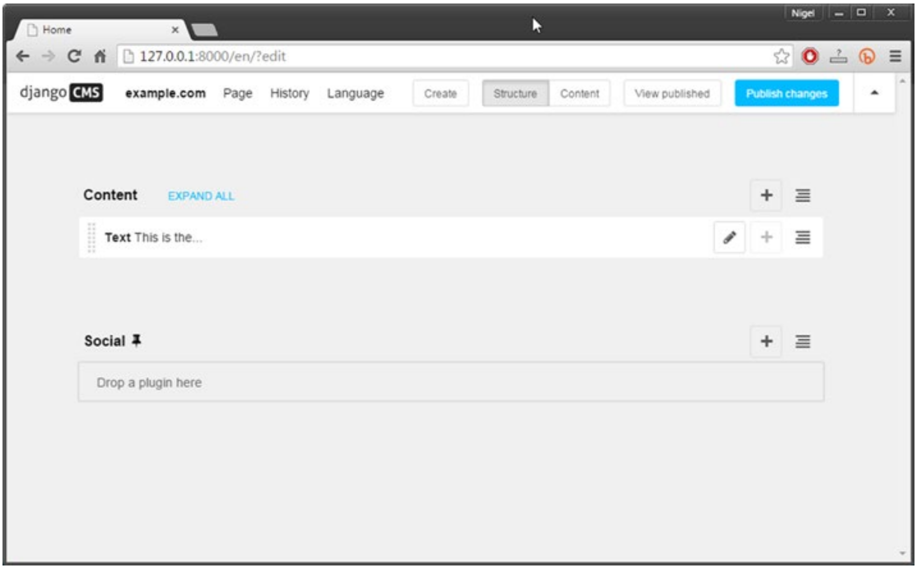


Figure 6-1. *An empty template in structural view*

The CONTENT and SOCIAL bars on the page correspond to the placeholders of the same name in your templates. These are the containers in which you place any plugins that you want to add to the page. For example, our content wizard has already added a Text plugin to the Content placeholder. If you look at the underlying HTML code (by choosing View Source) you will see that they are actually just HTML divs.

Each container has a contextual menu on the right side. If you click the plus (+) icon, a popup overlay (Figure 6-2) will appear in the center of your screen. Each item on the list is a *plugin*. For the remainder of this chapter, we will look at each of these plugins in turn and you will learn how to use them to create different kinds of content for your website.

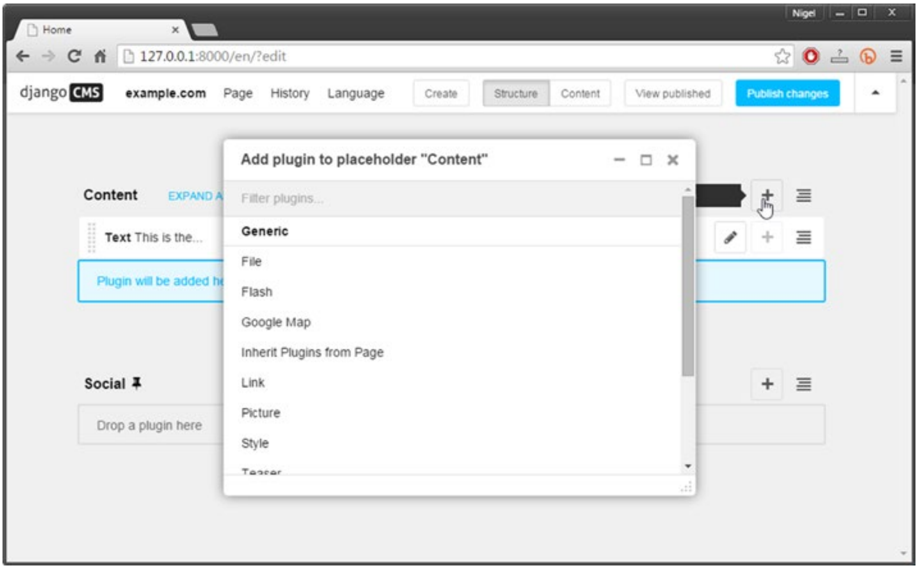


Figure 6-2. *A placeholder context menu*

Default Plugins

There are 11 default plugins installed by django CMS; together they provide all the basic functionality needed to present and format common web content like text, video and audio, images, and files. There are many more django CMS plugins available—from both the django CMS project team and third-parties—and you can easily add your own custom plugins by copying and enhancing one of the default plugins, or building your own from scratch. We will cover third-party plugins in the next chapter and building your own custom plugins in Chapter 8, but first let's look at the default plugins in more detail.

■ **Note** If you have used previous versions of django CMS, please note that placeholders and plugins are no longer managed from the administration backend; they must now be managed from the front end from within the structure view.

Installing the Default Plugins

This step is not necessary if you have followed the installation instructions in Chapter 2, however, it's worthwhile knowing how to install a default plugin if you have to reinstall, or want to use it on a custom site.

Installing a default plugin is very straightforward; just do the following:

1. Install the plugin using `pip`.
2. Add it to your `INSTALLED_APPS` in `settings.py`.

For example, you would install the File plugin with `pip install django-cms-file` and add `'djangocms_file',` to your `INSTALLED_APPS` to make in available in your application.

For instructions on installing all of the default plugins individually, see http://www.django-cms.org/en/addons/.

The Text Plugin

If you have ever used an online, browser-based text editor, the Text plugin will be familiar to you. If you haven't, don't worry because it is very straightforward and works like a simplified word processor. You will be using the Text plugin a lot, definitely more than any other plugin in the system, so it pays to really understand how it works. Select the Text plugin from the dropdown menu, and a window like the one in Figure 6-3 will pop up on to your screen.

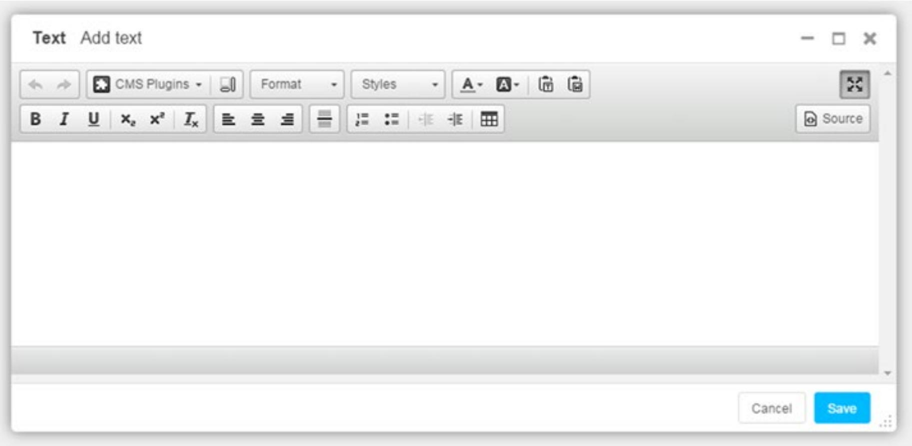


Figure 6-3. *The django CMS Text plugin*

The functions of the editor may seem basic compared to a word processor or advanced HTML editor, but in content creation for the web, more is not necessarily better. Table 6-1 lists each of the functions of the basic editor.

***Table 6-1.** Django CMS Text Plugin Toolbar Functions*

Icon	Function
	Undo/redo
	Insert file/link/picture or video. Each of these plugins behaves exactly the same way as when used directly on the page.
	Show blocks. Shows dotted outline of HTML blocks on the page. Useful for visualizing layout and detecting empty HTML tags.
	Paragraph format dropdown. Options include Heading, normal text, and other formats.
	Apply special text styles like code formatting, deleted text, and quotes.
	Select text foreground and background color.
	Paste Plain Text and Paste from Word. The latter cleans out Microsoft Work formatting tags.
	Text formats: Bold, italic, underline, subscript, and superscript.
	Clear formatting from selected text.
	Text alignment: left, center, or right.
	Insert a horizontal rule.
	Lists. Numbered, bullet. List indent and outdent.
	Insert table. This brings up another window where you can enter row, column, and table width and height information.
	Edit HTML source directly.

There are some limitations to the Text plugin—there is no WYSIWYG resizing of tables or embedded plugins (images, files and video), and any advanced HTML requires editing the source directly.

In practice, these problems are not as big as you might think.

For starters, using fixed-width columns in tables is almost always a bad thing, because they won't scale and flow properly on small screens; and you should **never** use tables for layout. Media files can be previewed quickly by switching to the content view, and the plugin system makes layouts really simple.

The most important concept to understand is that you don't need to lay out the whole page with one single Text plugin. This is probably the first mistake people new to django CMS make; especially if they have used other CMSes in the past. Let me illustrate by expanding on the quick example I introduced at the beginning of this chapter. Suppose you have a fairly standard page: banner at the top, followed by some introductory text, an image floated left or right and some more text floating alongside the image and continuing underneath. Generally in this situation, it is the image element and the banner that have a page author fiddling with HTML to get everything to look right.

In django CMS, you would achieve this by dropping in a Picture plugin for the banner, followed by a Text plugin with the introduction. Below this you would use a Multi Column plugin, splitting the percentage width to suit the image and placing the image in one column and the article text in the other. You then finish with another Text plugin below containing the rest of the article (the additional plugins mentioned in this example will be explained in full shortly).

Note that this is not a great example; normally, you should not have any trouble with text flow around an image for the main article, so you would not go for such an elaborate layout, but it *is* really important to understand that you have this flexibility if you have more demanding layout requirements.

And finally, there is nothing stopping you from using an external HTML editor and pasting the source straight into the text plugin. The Paste From Word button even allows you to use Word as your HTML editor (but please don't).

The Link Plugin

The Link plugin (Figure 6-4) inserts a link into the page content. It can also be used to insert links inside the Text plugin and the Multi Columns plugin. There are five link types available—Link (URL), Link to a Page on the Site, Link to an Anchor, Email Link, and Telephone Link.

■ **Note** The Link plugin does not throw an error if you enter multiple links, so make sure you fill out only one of the link fields.

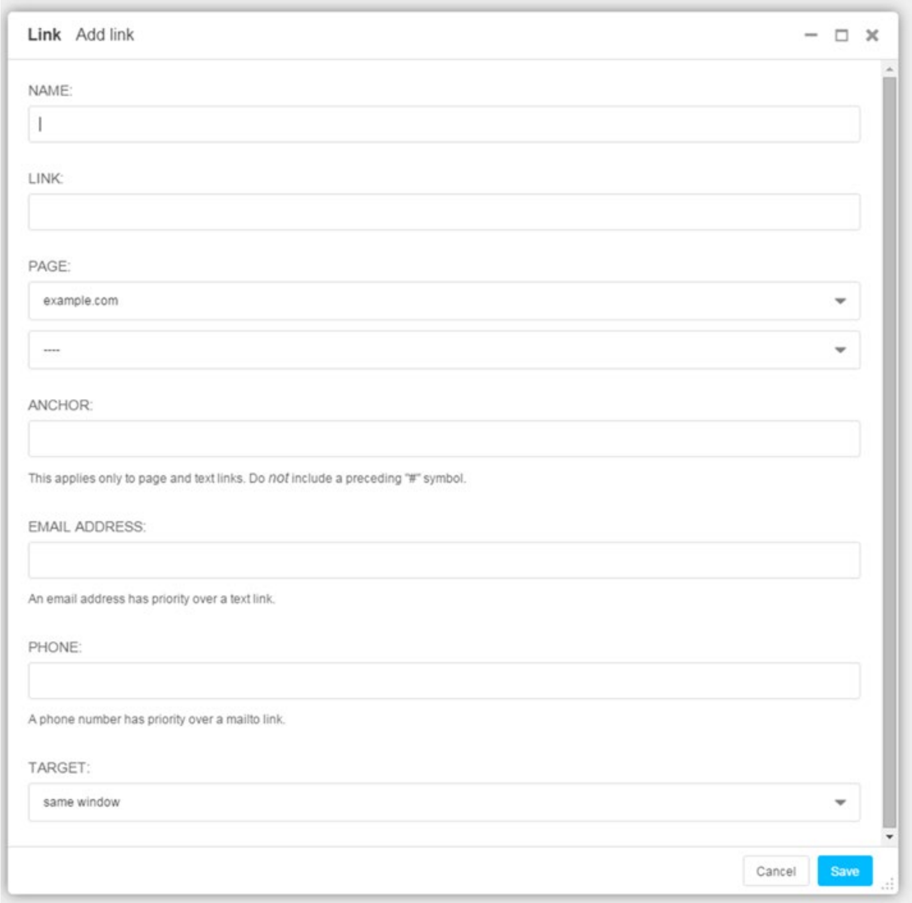


Figure 6-4. *The Link plugin*

There are a few things to note about the Link plugin:

- When you enter the URL into the Link field, it must have the http:// or http:// prefix, raw URLs will cause an error.
- You need to create the anchors in the page yourself by choosing Edit Source and entering <a name="anchor_name" /> (replace anchor_name with your anchor name). Then link the Anchor field to your anchor name. There is a hack for the Text plugin to fix this, but it is a bit advanced for this point in the book. I have included it in the accompanying sidebar.
- The Telephone Link will insert a tel://... link into your page. Depending on what software you have on your computer or tablet, this may launch another application. On a smartphone, it will simply call the number.

ADDING "INSERT ANCHOR" TO THE TEXT PLUGIN

Adding anchors to your source code is quite simple to do and is suitable for most content-creation jobs; however, if you find you are entering a lot of anchors in your pages, a tool to add them in the Text plugin is going to be useful.

Luckily, the Text plugin toolbar is based on a JavaScript HTML editor called CKEditor. What is really cool about CKEditor is that it already has an insert anchor tool, and all we have to do is enable it.

Open up your `settings.py` and add this to the bottom of the file:

```
CKEDITOR_SETTINGS = {
 'toolbar':[
        ['Undo', 'Redo'],
        ['cmsplugins', '-', 'ShowBlocks'],
        ['Format', 'Styles'],
        ['TextColor', 'BGColor', '-', 'PasteText', 'PasteFromWord'],
        ['Maximize', ''],
        '/',
        ['Bold', 'Italic', 'Underline', '-', 'Subscript',
        'Superscript', '-', 'RemoveFormat'],
        ['JustifyLeft', 'JustifyCenter', 'JustifyRight'],
        ['HorizontalRule'],
         ['Anchor'],
        ['NumberedList', 'BulletedList', '-', 'Outdent', 'Indent',
        '-', 'Table'],
        ['Source']
        ],
}
```

Reload your page (you may have to clear your browser cache and restart the development server), and you will have a nice flag icon (AKA the Insert Anchor tool) next to your horizontal rule button.

There are many other things you can do to modify the editor if you are feeling adventurous, but make sure you back up your files first!

The Picture Plugin

The Picture plugin (Figure 6-5) inserts an image into your page. It can also be used inside the Text plugin and the Multi Columns plugin. The Picture plugin does not allow image resizing on uploading, nor does it do any automatic thumbnail conversions; so it's important that your image is the right size before you upload.

■ **Tip** To create a thumbnail image that you can click to display the full sized image, save the full size image to `myBlogProject\media\` and then set Link: text on your thumbnail to `"../media/<full size image filename>"`. There is a plugin that we will install a bit later that can help make this process easier, but first it helps to learn how to achieve this outcome with the default settings.

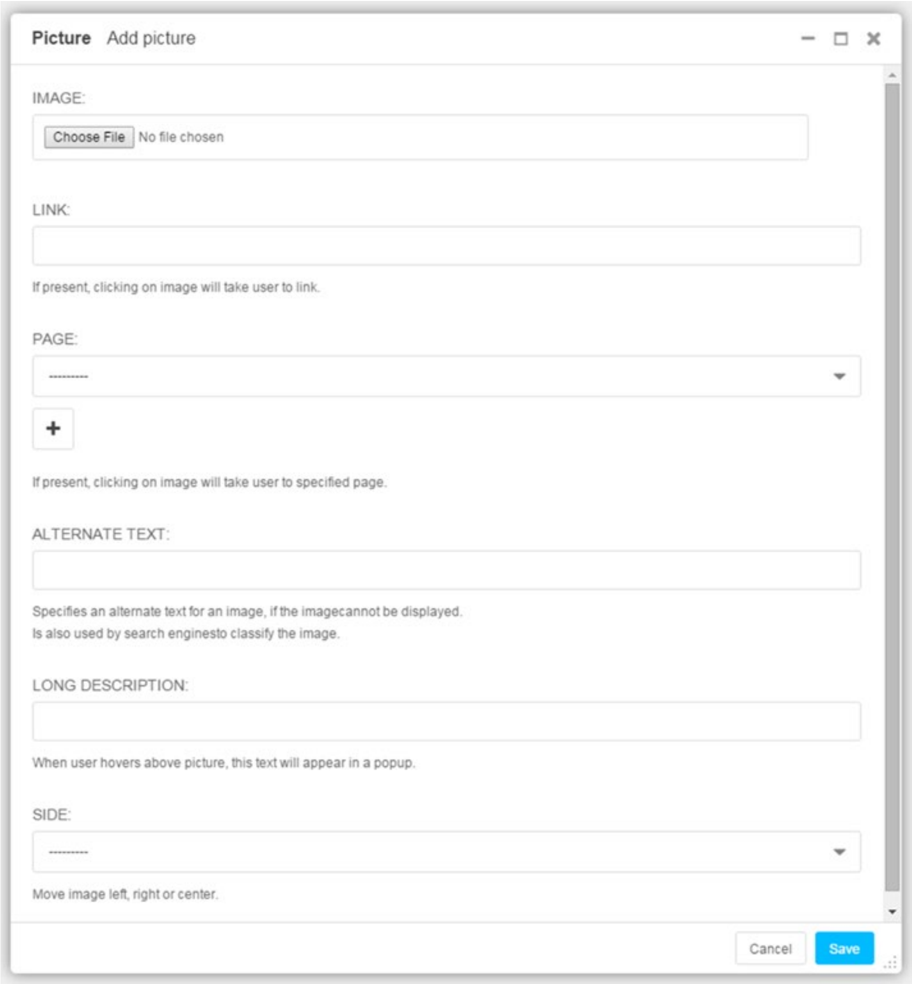


Figure 6-5. *The Picture plugin*

The File Plugin

The File plugin (Figure 6-6) inserts a file into your page. Like the Picture plugin, it can also be used inside the Text and the Multi Columns plugins. It is a very simple tool—you just need to pick the file for upload and select a title (which will be used as your anchor text).

The File plugin provides only the most rudimentary file embedding capabilities. For more complex sites where you will be embedding files regularly, it's recommended that you install and use the Django-Filer plugin, which you will learn about in the next chapter.



Figure 6-6. *The File plugin*

The Video Plugin

The Video plugin (Figure 6-7) embeds a video into your page. It can also be used inside the Text and Multi Columns plugins. The embedded video can be either a file on your site or an embedded link to a video on an external site like YouTube or Vimeo. Here are a few things to keep in mind when using the Video plugin:

- If you want to embed your own videos, you may have to play with the encoding to get the results you want. Generally, any common video container (.avi,.mp4, .m4v, .mkv) will work if the H.264 codec has been used. There is a wealth of information online about how to transcode video files if you are having problems.
- Width and Height are mandatory fields. If you are using a preview image, set these to the image dimensions, if you are linking to an external video, it is a good idea to maintain the same aspect ratio as the original video, such as 400 wide and 300 high for a 4:3 video.
- The Show Colors tab at the bottom of the Video plugin configuration screen allows you to customize a range of color settings for the embedded video.

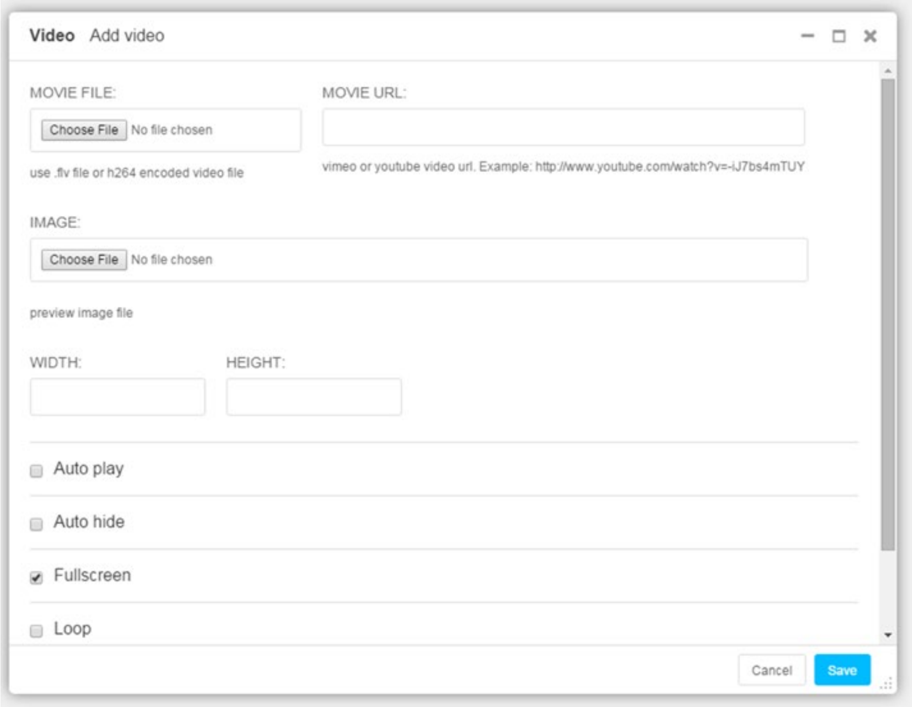


Figure 6-7. *The Video plugin*

■ **Tip** If you want to apply the custom settings (including color settings) to all the videos on your site, they can be added to your `settings.py` file. The available settings are as follows:

```
VIDEO_AUTOPLAY ((default: False)

VIDEO_AUTOHIDE (default: False)

VIDEO_FULLSCREEN (default: True)

VIDEO_LOOP (default: False)

VIDEO_AUTOPLAY (default: False)

VIDEO_BG_COLOR (default: "000000")

VIDEO_TEXT_COLOR (default: "FFFFFF")

VIDEO_SEEKBAR_COLOR (default: "13ABEC")

VIDEO_SEEKBARBG_COLOR (default: "333333")
```

`VIDEO_LOADINGBAR_COLOR (default: "828282")`

`VIDEO_BUTTON_OUT_COLOR (default: "333333")`

`VIDEO_BUTTON_OVER_COLOR (default: "000000")`

`VIDEO_BUTTON_HIGHLIGHT_COLOR (default: "FFFFFF")`

The Multi Columns Plugin

The Multi Columns plugin (Figure 6-8) allows you to format your page content in columns. It's not just useful for multiple-column text; it's a container for any of the other default django CMS plugins, as well as for compatible third-party and custom plugins. You can even nest Multi Columns plugins inside each other.

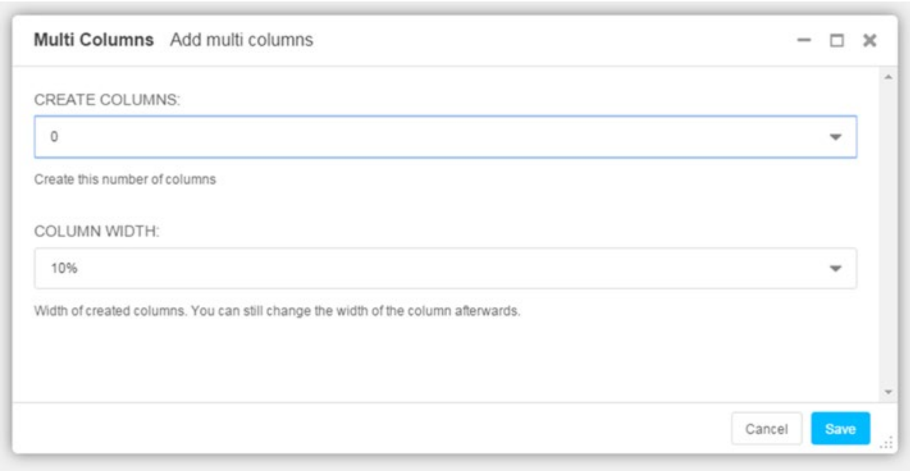


Figure 6-8. *Multi Columns plugin*

Using the Multi Columns plugin is very straightforward—choose how many columns you want and the default column width. Both settings can be changed easily later.

The Style Plugin

The Style plugin (Figure 6-9) allows you to add custom styling to page elements placed inside it. For example, if you want to apply custom styles to a Text plugin, you place the Text plugin inside the Style plugin.

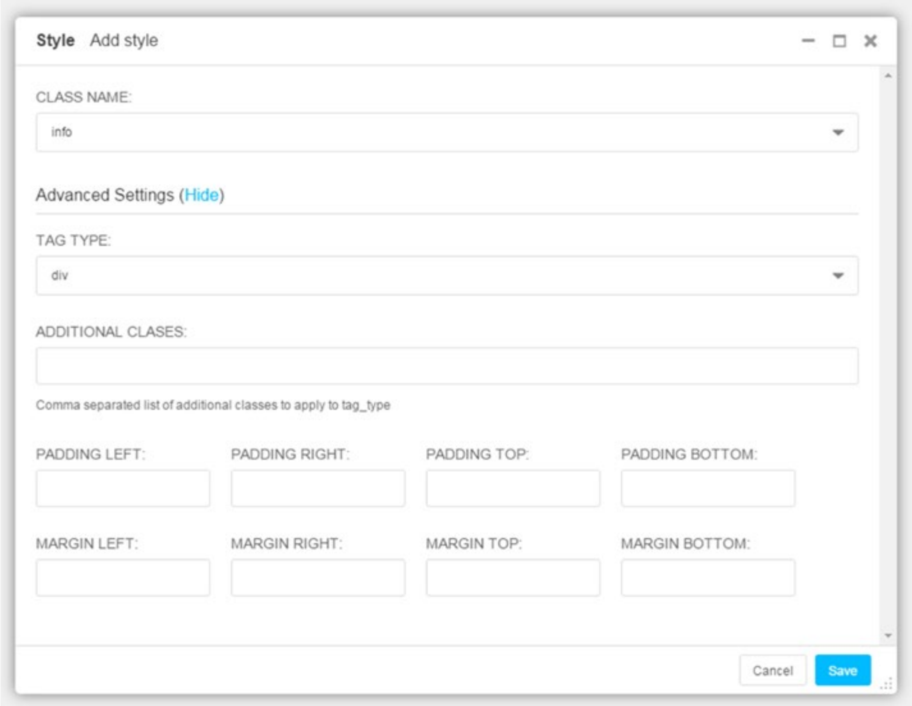


Figure 6-9. *The Style plugin*

Using the Style Plugin is straightforward; just select a class name from the list. This will apply the class name to the HTML div. You then set the style formatting in your custom CSS file. For example, if you select the `info` class, it will apply the styling of the `.info` class from your CSS file.

There is nothing special about the default classes; you can add more classes or replace them entirely by modifying your `settings.py` file. For example, to add a new class `mystyle` to the dropdown, add the following to your settings:

```
CMS_STYLE_NAMES = (
  ('info', ("info")),
  ('new', ("new")),
  ('hint', ("hint")),
  ('mystyle', ("mystyle")),
)
```

There are also a number of advanced settings you can add to the Style plugin, including these:

- Using HTML 5 article or section tag instead of the default div tag
- Applying additional classes
- Setting padding and margins

■ **Tip** You can add Bootstrap classes to the additional classes list. This is really useful if you want to apply Bootstrap contextual classes (like `.warning`, or `.danger`) to content.

The Teaser Plugin

The Teaser plugin (Figure 6-10) displays an image with a heading above and a block of plain text below it. It's best used inside a Multi Columns tag to float it left or right inside your page text, but like everything else in django CMS, its uses are limited only by your imagination.

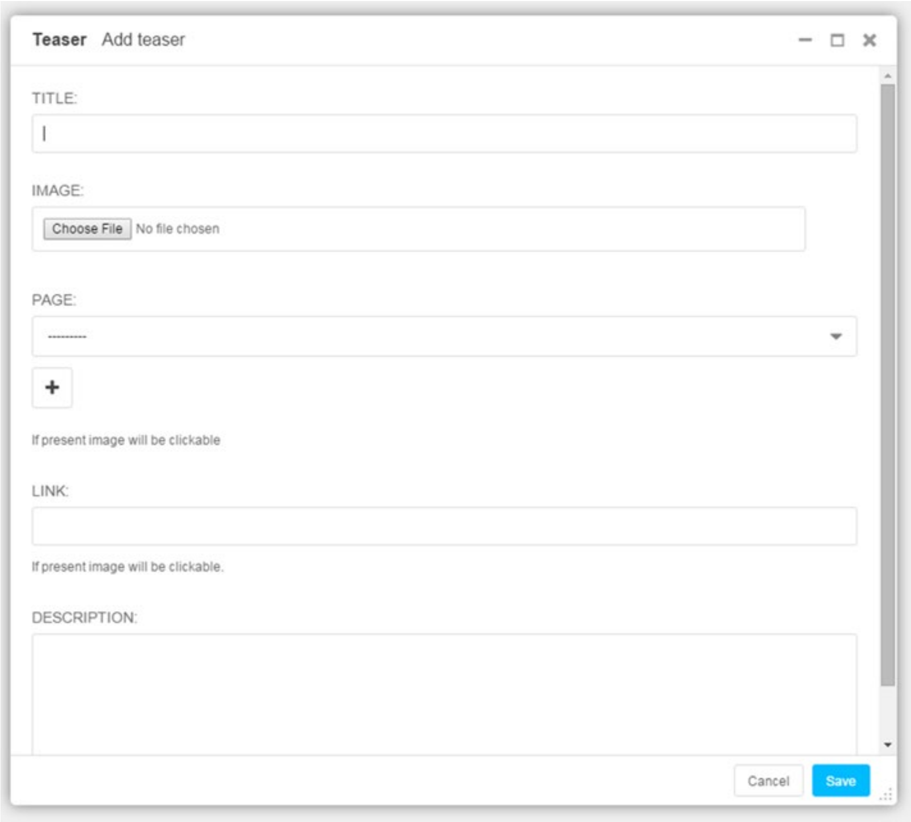


Figure 6-10. *The Teaser plugin*

Because the description is plain text and will render the same as your body text, it is also worth wrapping your Teaser plugin inside a Style plugin to apply custom styles to the teaser.

The Flash Plugin

The Flash Plugin (Figure 6-11) is another very simple plugin—just select your Flash (swf) file and specify a width and height.

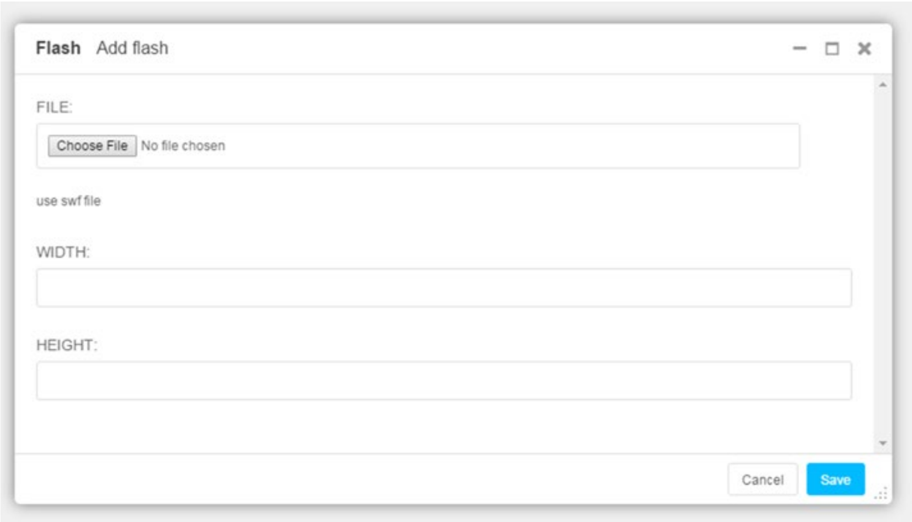


Figure 6-11. *The Flash plugin*

■ **Warning** Because we are working on a mobile-friendly website, it should be noted that Flash files have lots of compatibility issues across several platforms. The general advice is not to use Flash unless you have a very good reason to do so.

The Google Map Plugin

The Google Map plugin (Figure 6-12) embeds a Google Maps map in your page. The settings are straightforward; however, there are a couple of things to note:

- Outside of the United States, you will have better luck displaying the correct map reference by entering `<city>, <state/province>, <country>` in the City: field.
- The Latitude and Longitude settings are not direct map references; they are for fine-tuning the exact placement of your map pin.

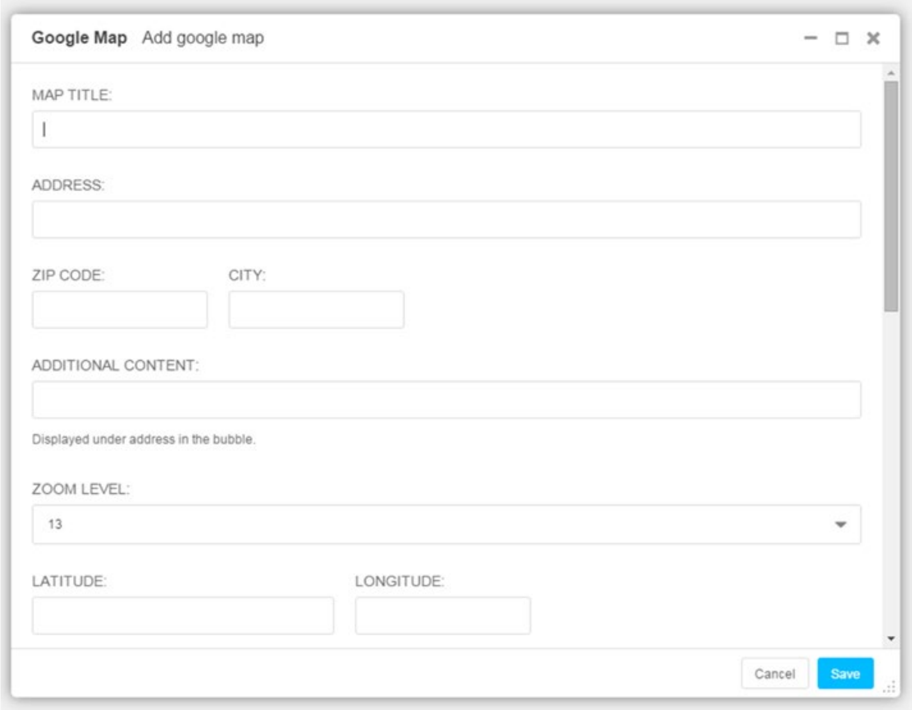


Figure 6-12. *The Google Map plugin*

The Create Alias Plugin

The Create Alias plugin creates an alias for the selected plugin, which allows it to be used on any page of your website, and any updates made to one instance of the aliased plugin will be automatically applied to all the other instances. It is not designed be used on its own, rather as a way to copy a plugin to multiple pages on your site and have the plugin content update automatically everywhere when it is changed. Figure 6-13a and 6-13b illustrate the alias process.

In Figure 6-13a, I have used Create Alias to create an alias for a Text plugin on the page.

Creating a linked copy of the plugin is as simple as pasting the alias into a placeholder or plugin on the page, as shown in Figure 6-13b. Note here that I am pasting onto the same page for the purposes of demonstration; in a real website, you would be pasting this on to a different page.

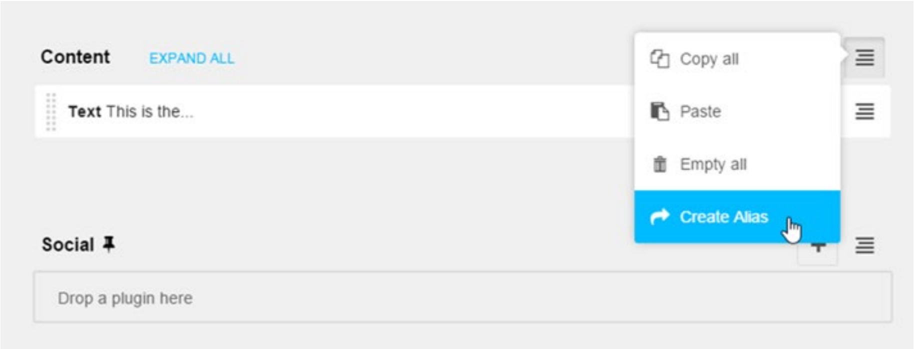


Figure 6-13(a). *Copying a plugin with Create Alias*

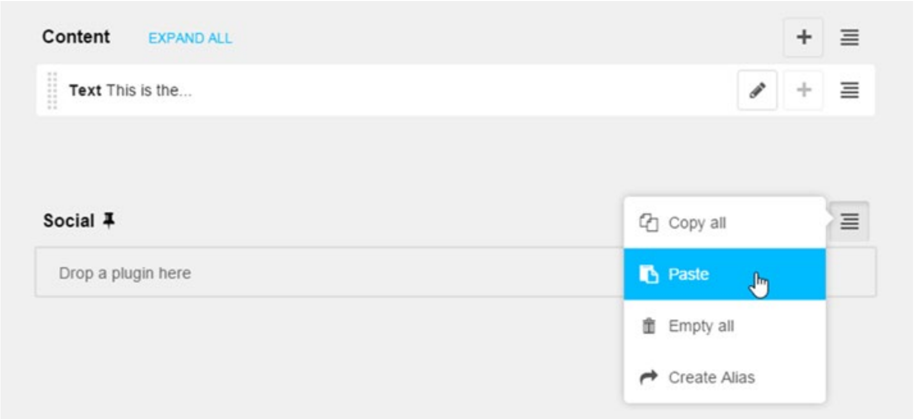


Figure 6-13(b). *Adding the aliased plugin to a page*

Summary

In this chapter we have looked briefly at the functionality provided by the built-in plugins in django CMS. We will putting each of these plugins to use in Chapter 8.

In the next chapter we will be adding to the functionality of the basic plugins by adding some of the great plugins that other developers have made for django CMS.

CHAPTER 7

Advanced Plugins

In the previous chapter, we covered all the default plugins that come preinstalled in django CMS. For simpler websites where content does not change often, these are more than enough to create and manage a website.

However, once you have a website with more complex content and the need to publish and update content more regularly, there are a number of third-party plugins available that will make creating and managing content on your django CMS website much easier.

This chapter introduces a number of third-party applications that extend and enhance the plugin functionality of django CMS. In the next chapter we will start putting them to use building content for your blog site.

Easy Thumbnails

If you remember from the last chapter, in order to create a thumbnail with the default Image plugin, you must manually resize the source image to create a thumbnail and then link to the source image for a full resolution view.

Because this is a very common task in a CMS, a number of applications have been developed for creating thumbnails; Easy Thumbnails is such an application. Originally created for Django, Easy Thumbnails also works very well with django CMS. It is designed to create thumbnails dynamically—automatically creating thumbnails for images you place in your page based on a simple template.

Note Easy Thumbnails is best suited to smaller sites that do not require sophisticated media handling. If your site is more complex, you are better off using CMSplugin-filer, which has Easy Thumbnails built in. The latest version of CMSplugin filer can be downloaded from `https://github.com/stefanfoulis/cmsplugin-filer`.

To use Easy Thumbnails, you must first install it using pip:

```
$ pip install easy-thumbnails
```

Once Easy Thumbnails has been installed, we need to add it to our project. First, add it to your installed apps in settings.py:

```
INSTALLED_APPS = (
    ...
    'easy_thumbnails',
)
```

Then run the migrate command to add the necessary tables to your project:

```
$ python manage.py migrate easy_thumbnails
```

Once Easy Thumbnails has been installed into your project, it requires a template file to be able to correctly resize your images and create the thumbnail files. A simple template file is shown in Listing 7-1. To enable the template in your project, name the file picture.html and save it to \myBlog\templates\cms\plugins. (You will need to create the new folders in your templates directory.)

Listing 7-1. The Easy Thumbnails Template File (picture.html)

```
{% load thumbnail %}

{% if link %}<a href="{{ link }}">{% endif %}
   {% if placeholder == "content" %}
    <img src="{% thumbnail picture.image 300x600 %}"
{% if picture.alt %} alt="{{ picture.alt }}"{% endif %}
{% if picture.longdesc %} title="{{ picture.longdesc }}"{% endif %} />
   {% else %}
     {% if placeholder == "authorbio" %}
    <img src="{% thumbnail picture.image 50x50 crop %}"
{% if picture.alt %} alt="{{ picture.alt }}"{% endif %}
{% if picture.longdesc %} title="{{ picture.longdesc }}"{% endif %} />
      {% endif %}
    {% endif %}
{% if link %}</a>{% endif %}
```

This template is very straightforward. To see what it is doing, let's step through the code:

- We first need to load the custom thumbnail tags into the template with `{% load thumbnail %}`.
- The outer `if/else` statement selects the correct thumbnail sizing depending on the placeholder in your django CMS page template. We created the `content` placeholder in Chapter 2. Note that for this to actually work, you would need to modify your base (or a child template) to include the `authorbio` placeholder.
- The thumbnail creation magic is performed by the `{% thumbnail picture.image ... %}` tag. It sets the size of the thumbnail, and in the case of the author avatar, crops the image.
- The additional `if/else` tags create an anchor if there is a link attached to the image and `alt` and `title` tags for the image if those elements exist.

A final note on configuring Easy Thumbnails. If you plan to reuse your app, or want to maintain consistency and flexibility across your site, it is better to use an alias than to hard-code thumbnail size, for example, by adding the following to your `settings.py`:

```
THUMBNAIL_ALIASES = {
    '': {
        'avatar': {'size': (50, 50), 'crop': True},
    },
}
```

Also modify your template so that this:

```
<img src="{% thumbnail picture.image 50x50 crop %}"
```

becomes this:

```
<img src="{% thumbnail picture.image avatar %}"
```

You now have the freedom to copy the template to multiple projects as well as set the image parameters for all templates that use the `avatar` alias, without having to modify multiple files.

This has been only a brief introduction to Easy Thumbnails; for a complete description and details of all relevant configuration options and settings, please refer to the Easy Thumbnails documentation [1].

[1] http://easy-thumbnails.readthedocs.org/en/2.1/

CMSplugin-filer

CMSplugin-filer adds Django-filer, a file manager for Django, to django CMS. It provides a number of functions you would expect from a file management application:

- Create, rename and delete file folders.
- Upload, rename and delete files of any type (although you will most likely only be managing common media files like images, videos, PDF and document files.
- Organize media files on your server.
- Apply file permissions and secure downloads (experimental features at the time of writing).

To install CMSplugin-filer, we first install the application files with `pip`:

```
pip install cmsplugin-filer
```

To ensure that CMSplugin-filer installs correctly, we must first add Django-filer to our project. Add the following to your `settings.py`:

```
INSTALLED_APPS = (
. . .
'filer',
'easy_thumbnails',
)
```

Then you need to install the application tables into your project. To do this, run two commands:

```
python manage.py makemigrations
python manage.py migrate
```

The first command ensures that all our migrations are synced, and the second command performs the actual migration. Now it's time to install the CMSplugin-filer application files. First we must make some changes to `settings.py`:

```
INSTALLED_APPS = (

    . . .

    'cmsplugin_filer_file',
    'cmsplugin_filer_folder',
    'cmsplugin_filer_link',
```

```
    'cmsplugin_filer_image',
    'cmsplugin_filer_teaser',
    'cmsplugin_filer_video',
)

. . .

MIGRATION_MODULES = {

    . . .

    'cmsplugin_filer_file': 'cmsplugin_filer_file.migrations_django',
    'cmsplugin_filer_folder': 'cmsplugin_filer_folder.migrations_django',
    'cmsplugin_filer_link': 'cmsplugin_filer_link.migrations_django',
    'cmsplugin_filer_image': 'cmsplugin_filer_image.migrations_django',
    'cmsplugin_filer_teaser': 'cmsplugin_filer_teaser.migrations_django',
    'cmsplugin_filer_video': 'cmsplugin_filer_video.migrations_django'
}
```

Then we just need to run `migrate` again to ensure that we have all the necessary tables added to the project:

```
python manage.py migrate
```

If the installation is successful, you will have six new plugin options added to your page structure editor (Figure 7-1):

- File*
- Folder
- Image*
- Link*
- Teaser
- Video*

*These plugins are also added to the Text plugin.

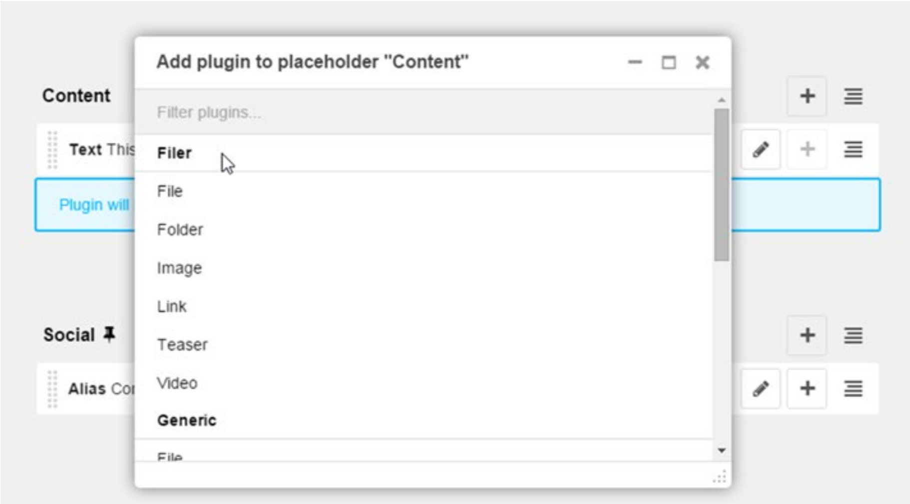


Figure 7-1. *Plugins added by CMSplugin-filer*

By the way, once you have installed CMSplugin-filer, you are unlikely to use the corresponding default plugins again. To reduce the number of options in your plugin dropdowns, you just need to remove the duplicates. You can do this easily by commenting out the duplicates in your `settings.py`:

```
INSTALLED_APPS = (

    . . .

    'djangocms_column',
    #'djangocms_file',
    'djangocms_flash',
    'djangocms_googlemap',
    'djangocms_inherit',
    #'djangocms_link',
    #'djangocms_picture',
    #'djangocms_teaser',
    #'djangocms_video',

   . . .

)
```

We will be using each of the filer plugins in a practical example in the next chapter. The following introduction to each plugin is provided for easy reference. We will start with the file manager, which is common to all the plugins, and then discuss each of the plugins in turn.

File Manager

Common to all six of the new CMSplugin-filer plugins is the file manager (Figure 7-2). The file manager is opened from within each plugin by clicking the search icon ( ). To select a file, click the select icon ( ) next to the file you wish to insert.

With the file manager you have the following options:

- **New Folder.** Adds a new folder under the selected folder. I have added a new folder called `siteimages` in this example.
- **Upload.** Lets you upload a new file from your computer to the selected folder. When you first upload a file it is added to the clipboard on the right of the screen, where you can either add it to the current directory or discard the upload if you change your mind.
- **Search.** Searches all folders for a file.
- **Change.** Lets you change file attributes and metadata. The options available will depend on the file type being uploaded. You'll learn more about the Change option when we look at the individual plugins.

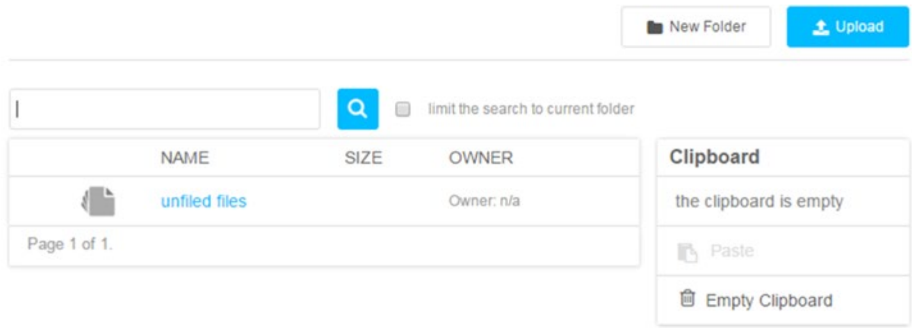


Figure 7-2. *The CMSplugin-filer file manager*

■ **Note** CMSplugin-filer installs the complete Django-filer application, which provides an effective media manager that is accessible not only from the front end, as in these examples, but also via the administrative back end. We will be covering more of the admin back end in the next chapter.

The File Plugin

The filer File plugin is very simple (Figure 7-3); just enter your title (which will become the file anchor text) and the file you want to embed.



***Figure 7-3.** The CMSplugin-filer File plugin*

The Folder Plugin

The Folder plugin (Figure 7-4) renders the contents of a folder as either a bulleted list or a slideshow. It is most useful for creating image slideshows; however, it is also a convenient tool for listing the contents of a directory on a web page.

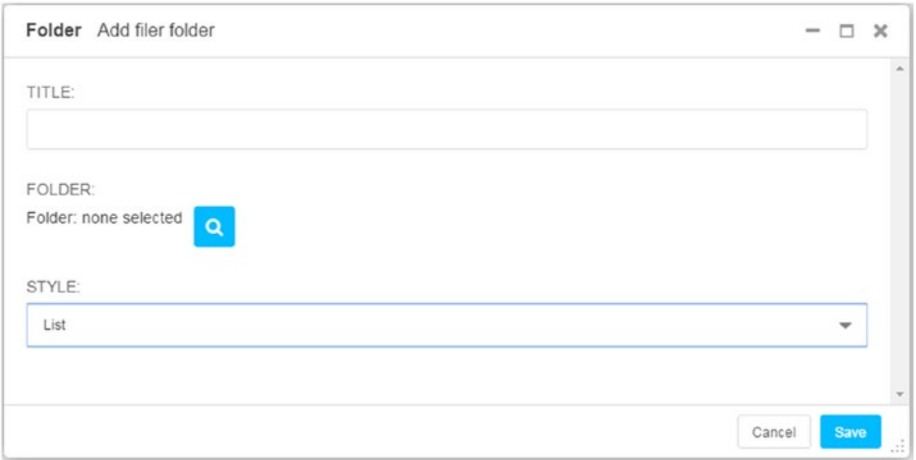


***Figure 7-4.** The Folder plugin*

Configuration of the folder plugin is simple; just enter a title for the plugin and select the folder you wish to attach. If you select the "list" style, each file in the folder will be listed on your page as an anchor with link to your file, if you select the "slideshow" style, all the images in your folder will display as a slideshow.

The Image Plugin

The Image plugin in CMSplugin-filer (Figure 7-5) provides a number of enhancements to the basic Picture plugin. Images can be added either via the File Manager, or linked directly using the Alternative Image URL field. You can also add a caption as well as the usual alternate text.

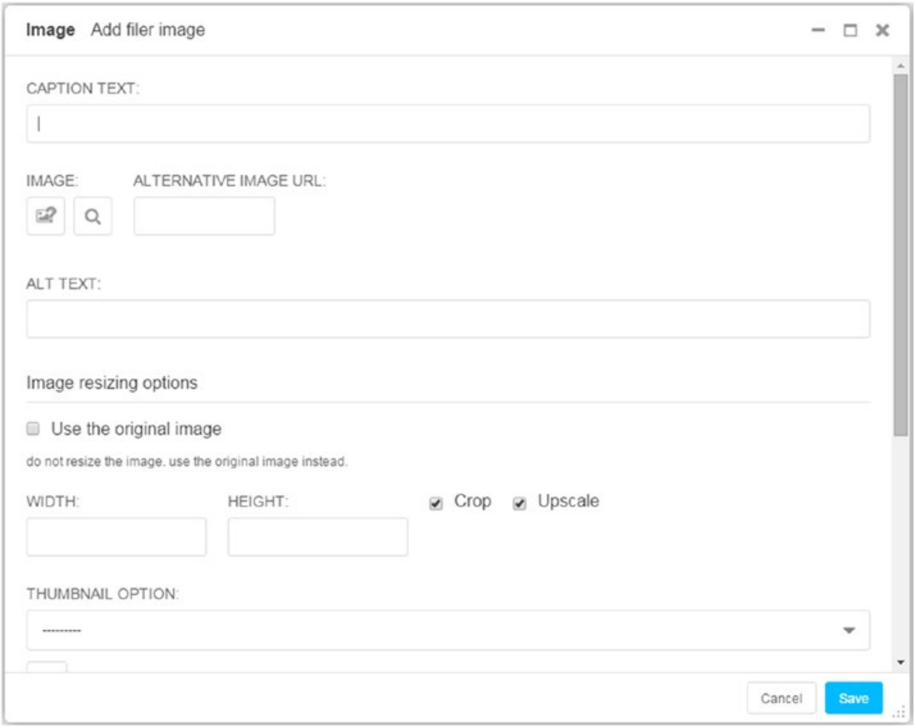


Figure 7-5. *The Image plugin*

The Image plugin also provides image scaling options and advanced linking (Figure 7-6). The image scaling and linking options are straightforward. Things to note:

- **Use the original image**. This overrides all other settings and simply displays the image without any resizing.
- **Width/Height**. You can specify either width or height to get proportional sizing, but for cropping to work, you must specify both.
- **Upscale**. Simply resizes the image without any resampling. Be aware that small images are likely to be low-quality if upscaled.
- **Thumbnail Option**. Allows the application of a predefined thumbnail size setting. Click the green plus (+) sign to add a new setting. Any new setting you add will be available to all images.
- **Use automatic scaling**. Allows the Image plugin to try to scale the image based on the context. Note that the image will be scaled differently depending on the placeholder it is contained within.
- **Image Alignment**. Does not apply any styles to the image, just an HTML class. It is up to you to provide the styles in a custom CSS file (`blog.css` in our case).
- **Link original image**. Opens the full-size image.
- **Description**. django CMS will render anything you put in this field as text inside a `span` tag. It is up to you to apply styling (see the accompanying note).

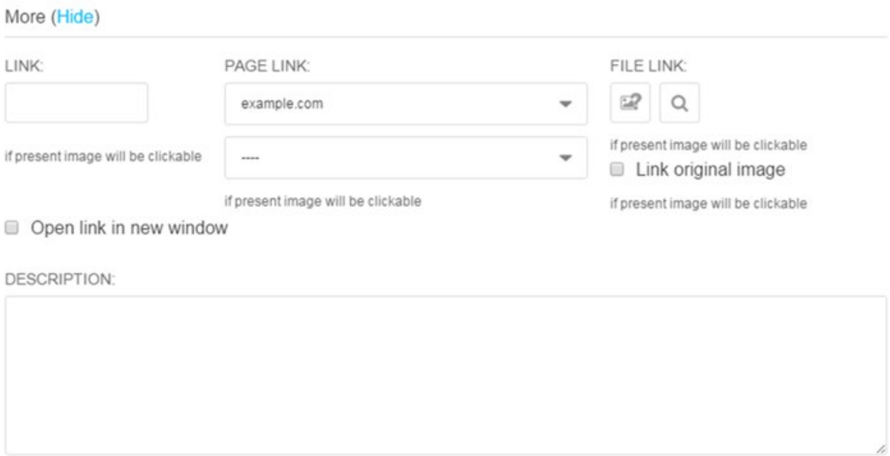


Figure 7-6. *Additional options for the Image plugin*

■ **Note** The image caption and description are added as HTML `span` tags, and so they will render inline with the image instead of sitting below it as you would expect. Fortunately, this is easy to fix. django CMS applies classes to the image caption and description (`title` and `desc` respectively). Add these classes to your `blog.css` file to display the caption and description below the image:

```
.title, .desc {
    display: block;
}
```

The Link Plugin

The CMSplugin-filer Link plugin (Figure 7-7) is very similar to the basic Link plugin, with the only major difference being the addition of the file manager. The Name field is the anchor text, and you are provided options for a direct link (URL), page, Mailto and file links. Like the basic version, this Link plugin does not throw an error if you fill out multiple link fields, so ensure that you enter information only in the field you want.

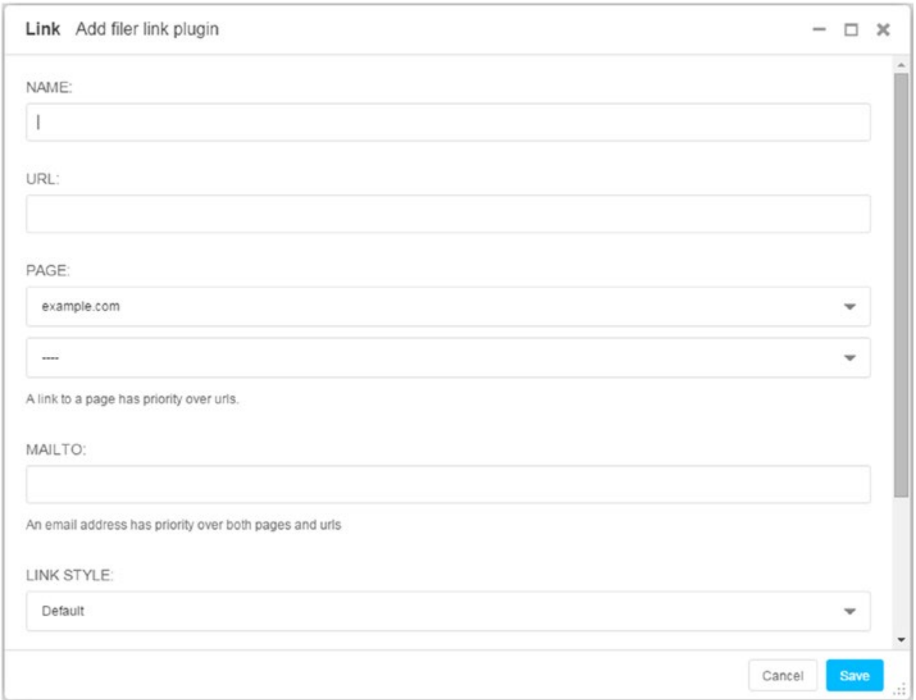


***Figure 7-7.** The Link plugin*

The Teaser Plugin

The Teaser plugin (Figure 7-8) in CMSplugin-filer also functions much the same as its basic version, with the addition of the file manager and image resizing options under the More link.

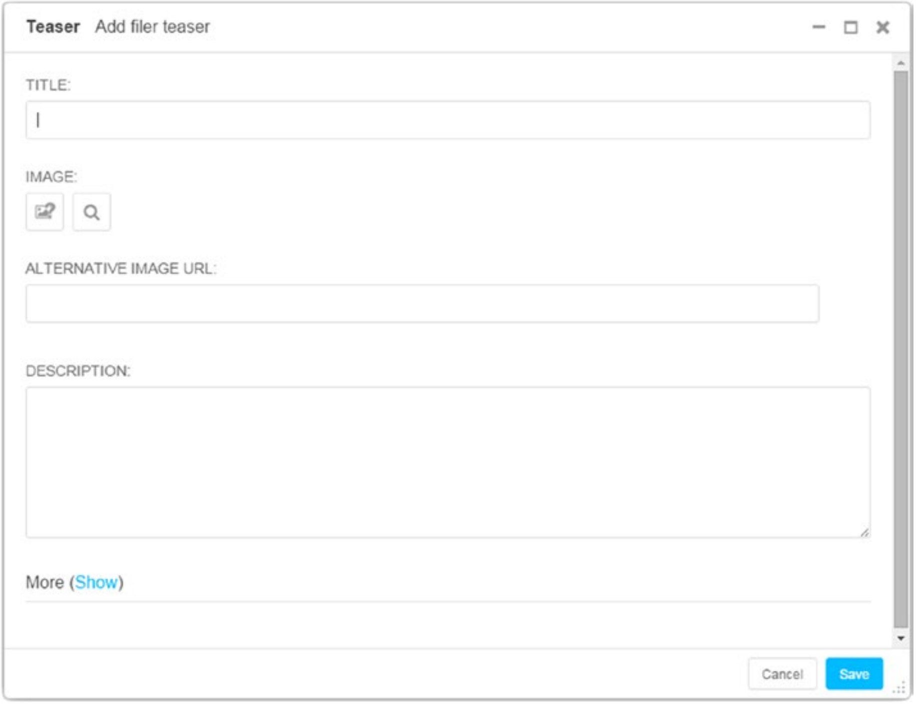


Figure 7-8. *The Teaser plugin*

■ **Note** The title renders as an HTML H2 tag and the description as a basic paragraph (the p tag) with no default HTML classes, so it's recommended that the CMSplugin-filer Teaser plugin be placed inside a Style plugin, just as with the basic version.

The Video Plugin

The CMSplugin-filer version of the Video plugin (Figure 7-9) is identical to the basic version, except for the addition of the file manager and a default size of 320×240.

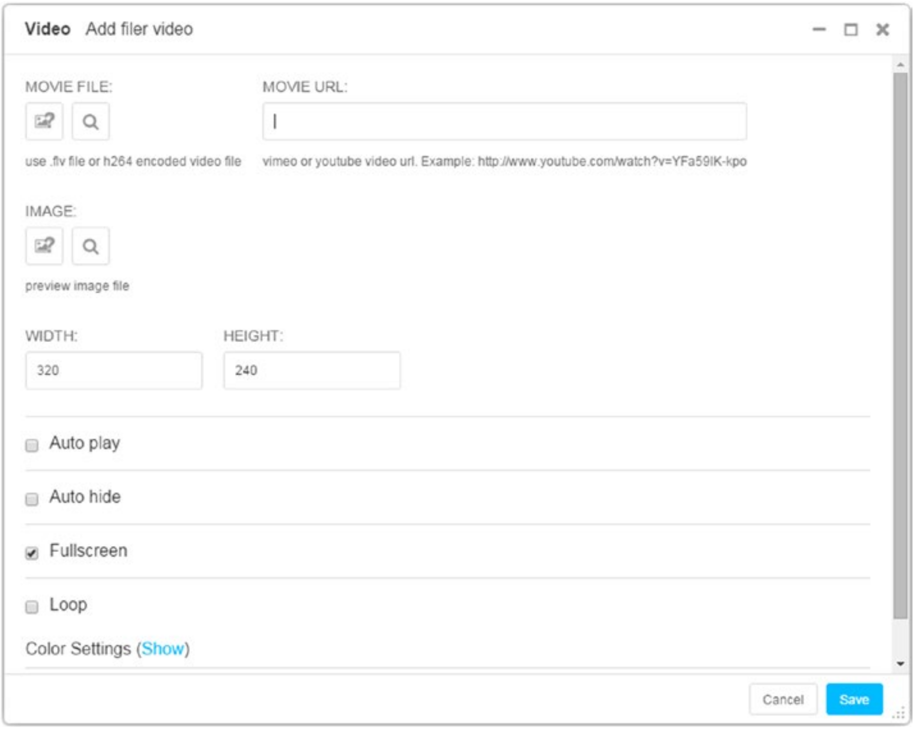


Figure 7-9. *The Video plugin*

djangocms-forms

djangocms-forms is a third-party form plugin for django CMS created by Mishbah Razzaque. It provides all of the form fields you will ever need, along with the ability to add CAPTCHA for spam filtering and saving of form submissions in the database; all without needing to write a single line of code. To use djangocms-forms in our project, first we need to install it:

```
pip install djangocms-forms
```

Then we need to add djangocms_forms to INSTALLED_APPS:

```
INSTALLED_APPS = (
    ...
    'djangocms_forms',
    ...
)
```

Finally, we sync the database:

```
python manage.py migrate
```

■ **Tip** So far, we have installed each of these plugins one at a time. It's probably a good time to note that all of the plugins (and apps) you wish to include in your website can be installed at once using a *requirements file*. Because requirements files are a feature of pip, not django CMS, I will not go into them here, but if you are curious, check out https://pip.readthedocs.org/en/1.1/requirements.html.

To use the djangocms-forms plugin in our application, we need to hook it into our project. First we need to add the form URLs to our project. Open \myblog\urls.py and add the following to your urlpatterns:

```
urlpatterns = patterns(
    ...
    url(r'^', include('djangocms_forms.urls')),
    ...
)
```

You should then be able to hook the form plugin into your page by going to Page ➤ Advanced Settings ➤ Application and selecting Forms from the dropdown. If the Forms option has not appeared in the dropdown, you may have to restart the development server.

Save the page settings and now when you edit your page, you will see the Form option under Generic. Select the plugin and you should be greeted by a popup that looks like Figure 7-10.

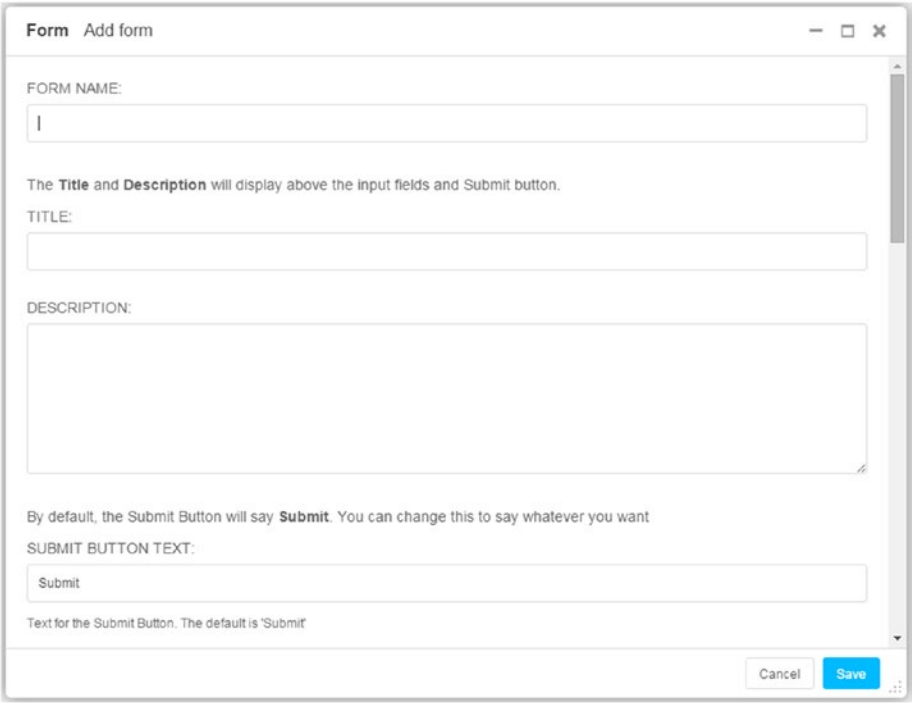


Figure 7-10. *The Djangocms-forms plugin (partial)*

Don't be daunted by all the options available; creating a detailed form is actually very easy, but first we have to take care of one thing.

Because we are in development, we do not have our email set up, and so we cannot test the form. Luckily, Django provides a file-based email back end designed specifically for the purpose of assisting developers test emails without having to configure an email server first. To get our email back end running, add the following to the end of your `settings.py`:

```
EMAIL_BACKEND = 'django.core.mail.backends.filebased.EmailBackend'
EMAIL_FILE_PATH = '/Users/<your username>/maildump'
```

You don't need to create the `maildump` folder; Django will do that for you the first time you try to send an email. When you do try to send email, you will find this folder contains files named something like `20151009-100625-98201520.log`. Inside this log file will be your complete message, including any mail headers and error messages. Very handy for testing.

Creating a Form

Figure 7-10 show a partial view of the popup that is displayed when you create a form. The configuration screen is quite long, but all the fields are straightforward:

- **Form Name.** The form name will displayed in your site's administration back end so you can tell which form was submitted. More on form administration in the next section.
- **Title.** This is the title that will be displayed in your page.
- **Description**. This will be displayed under the title in your page. You can use HTML tags here to format your code.
- **Submit Button Text**. Defaults to “submit”, but you can change this.
- **Post Submit Message**. The message that will be displayed on the page once the user has submitted the form. You can use HTML tags here to format your code. Tip: Use the Bootstrap class `alert alert-success` here to get a pretty formatted success box around your message.
- **Redirect?** Select a target if you wish to redirect the user after the form has been submitted.
- **Submissions Settings**. Sets the recipient(s), sender, and email subject. Also allows you to set a CAPTCHA. See `https://github.com/mishbahr/djangocms-forms` for more information on using CAPTCHA with djangocms-forms.

Adding Form Fields

Adding form fields is a simple as clicking the green plus (+) icon at the bottom of the form creation popup. This will expand the window, showing a configuration screen similar to Figure 7-11.

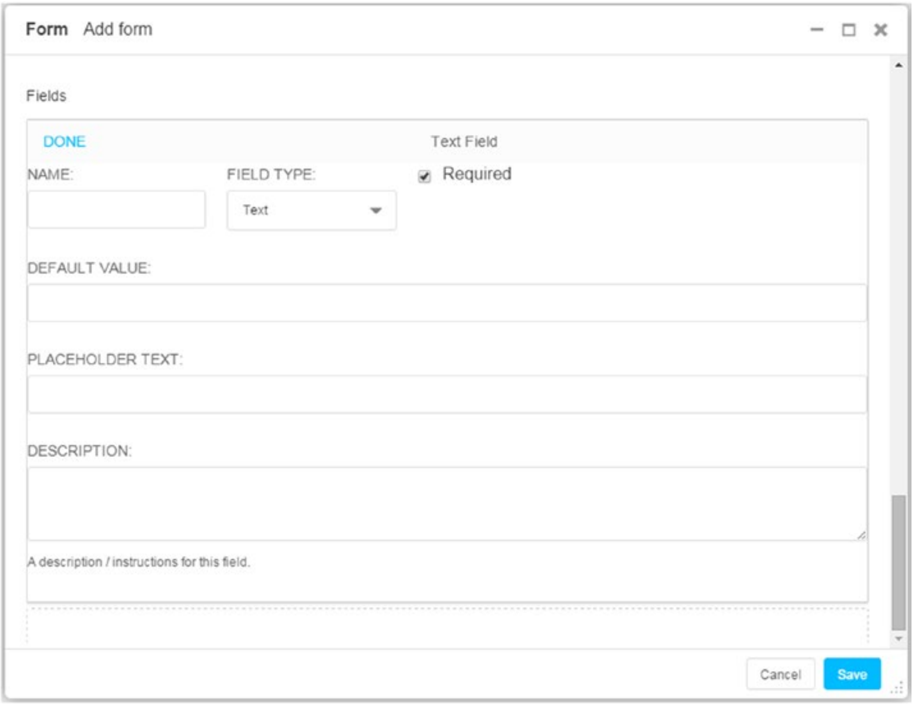


Figure 7-11. *Adding form fields*

The field editor has the following options:

- **Name**. Will be displayed to the left of the field in the form.
- **Field Type**. Data type of the field. Available options are:
  - Text, Text area, Email, Number, Phone, URL, Checkbox, Multi Checkbox, Dropdown, Radio, File Upload, Date, Time, Password, and Hidden.
  - Djangocms-form provides automatic validation for all relevant fields (such as Email and Number). All fields marked as “required” will also throw an error if they are empty.
- **Default Value**. Sets a prefilled value. Note that this will overwrite anything you enter in Placeholder Text.
- **Placeholder Text**. Grayed-out prompt text inside the field. This is used instead of a tooltip on most modern web forms.
- **Description**. Short description of the field that will appear directly below the field.

Form Administration

Selecting "Save to database" when configuring the form will save all the form fields into your project database. If you navigate to root (`example.com`) Administration, select Form Submissions from the sidebar, and expand to full screen, you will be greeted with a window similar to Figure 7-12.

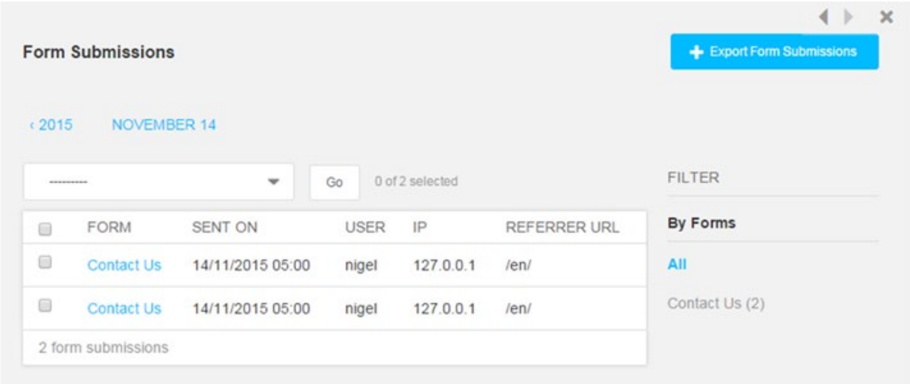


Figure 7-12. *Form Submissions in the Admin tool*

The Form Submission window allows you to delete submissions as well as export all the submitted forms to an external file. Available formats are CSV, JSON, YAML, and Microsoft Excel. Very useful for exporting form data for use in an external program or even importing to a mailing list manager.

Summary

We have covered only three plugins this chapter, Easy Thumbnails, CMSplugin-filer, and Djangocms-forms; there are so many plugins available for django CMS, with more being added each week, that there is not enough room to cover them all.

If you want to explore the full range of plugins available for django CMS on your own, a good place to start is the django CMS project site: `http://www.django-cms.org/en/addons/`.

In the next chapter we will put these three plugins, along with the basic plugins from Chapter 6, to good use in creating some content for our site.

CHAPTER 8

Authoring in django CMS

In the first seven chapters, we focused on designing our website templates and exploring the functionality of django CMS. In this chapter, we are going to put what we have learned into practice by creating the most important part of your website for your audience: content.

With front-end editing and flexible plugin architecture, django CMS provides powerful content formatting and editing capabilities for both administrators and site content creators.

The chapter starts with an overview of the django CMS toolbar and its functions. Then we will be using django CMS's built in plugins, as well as the third-party plugins we explored last chapter, to create three pages for your site—the home page, an about page, and a contact page. We will also create a couple of blog posts for you to use in later chapters once we have your blog application running.

Later in the chapter we will explore site administration in detail; in particular, we will dig deep into Django's `auth.user` model and will set user groups and permissions so that our website behaves like a professional CMS—with appropriate user, author, editor, and administrator roles and member-only content.

The django CMS Toolbar

The django CMS toolbar provides access to all the most important functions involved in site authoring.

Site Root (example.com)

The root menu for the site (Figure 8-1) provides quick links to page and user administration, as well as a link to the full administration back end. If you haven't changed the default site description in the admin interface, it will be called `example.com`. I will show you how to change this a little later in the chapter.

These menu options are only available to users with full administrative privileges. We will cover the administrative functions later in the chapter. The root menu also allows any registered user (not just administrators) to modify User Settings, including the default language in the admin interface, and to disable the toolbar. The latter is useful for previewing the website without the toolbar.

The root menu also provides access to the django CMS clipboard, which will contain either the latest copied plugin or the last plugin alias created.

■ **Tip** To re-enable the toolbar, return to edit mode (`/?edit`).

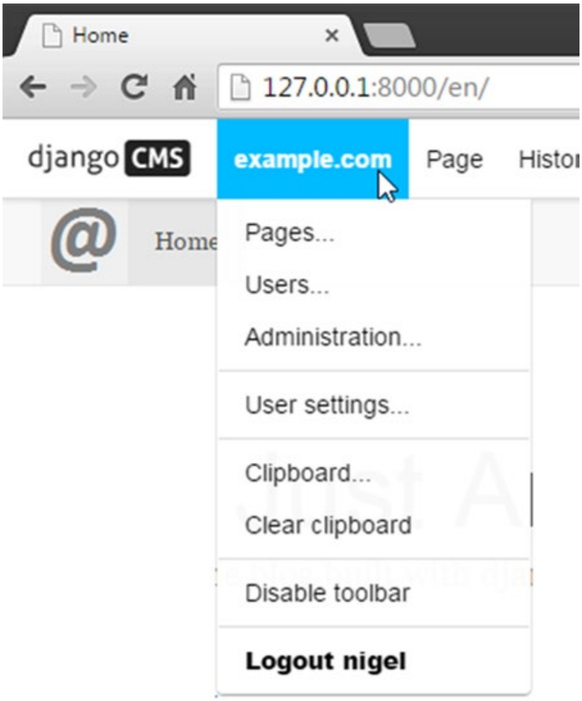


Figure 8-1. *The django CMS root menu*

Page

The Page menu (Figure 8-2) provides access to all the page management functions in django CMS.

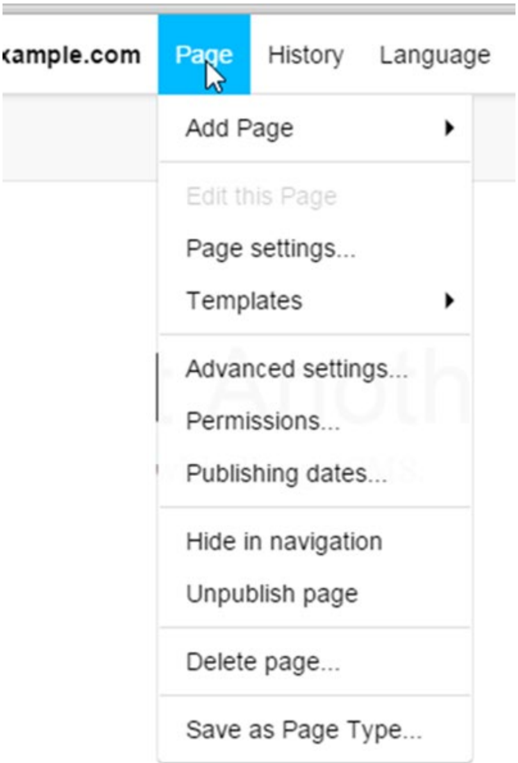


Figure 8-2. *The django CMS Page menu*

Add Page

There are three options for adding a page to django CMS:

- New Page
- New Sub Page
- Duplicate This Page

The page settings for these options are identical and are set via the popup overlay (Figure 8-3). Subpages are added under the currently selected page in your browser; however, this can be easily changed (see the page administration section later in this chapter). All these settings are self-explanatory, and you should have no trouble adding pages. Don't worry about trying to understand page type at this stage, as we will be discussing it shortly.

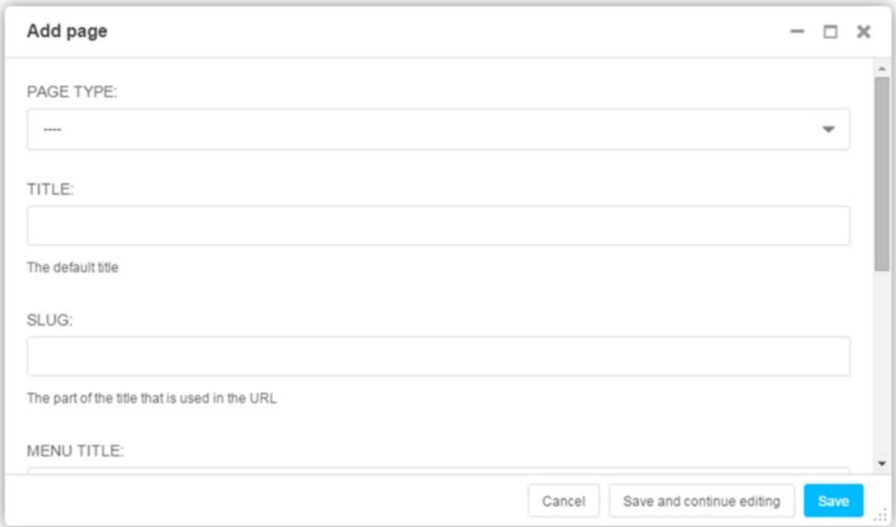


Figure 8-3. *Adding new pages*

django CMS provides a basic collection of page settings out of the box; however, the Page model is extensible. You will learn how to extend the Page model and add additional fields in Chapter 10.

Page Settings

The Page Settings menu item provides access to the same settings as Figure 8-3.

Templates

Select a template for the page. If you followed the instructions in Chapter 4, you should have two options here: Page and Blog Page with Sidebar.

Advances Settings

The Advanced settings menu allows you to set and modify a number of advanced page settings (Figure 8-4).

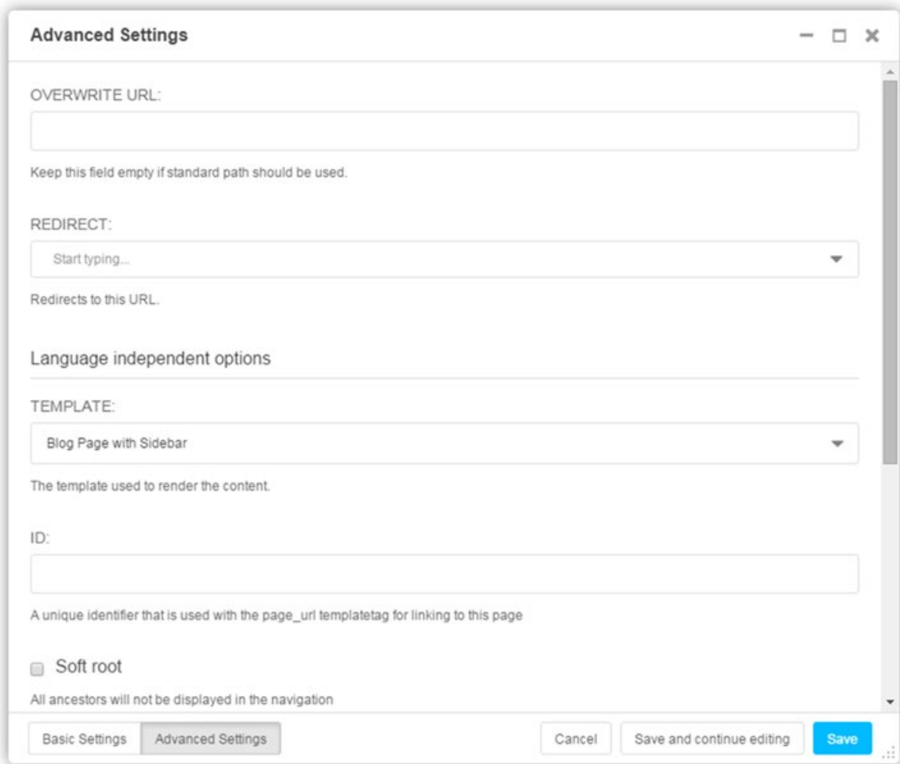


Figure 8-4. *Page advanced settings*

- **Overwrite URL**. Replaces the system-generated URL with a custom URL for the page.
- **Redirect**. Useful for redirecting old pages to a new URL. Can also be used to redirect to an external URL.
- **Template**. Has the same effect as changing the page template in Page settings.
- **ID**. This is an advanced option outside the scope of this book. It is used in template tags for advanced internal linking.
- **Soft root**. Creates a soft root at this page. Useful for creating submenus; pages checked as a soft root will not display any ancestors when attached to a menu. More on menus in the next chapter.
- **Attached Menu**. Custom menu attached to this page. More on menus in the next chapter.
- **Application**. Provides an apphook to a Django application. You will learn about apphooks in Chapter 10.
- **X Frame Options**. Sets whether the page can be embedded in another page. Used to prevent illegal framing of your webpages in other sites.

Permissions

Individual page permissions can be modified via the Permissions menu (Figure 8-5). Page permissions provide finer-grained control over pages than global user and group permissions.

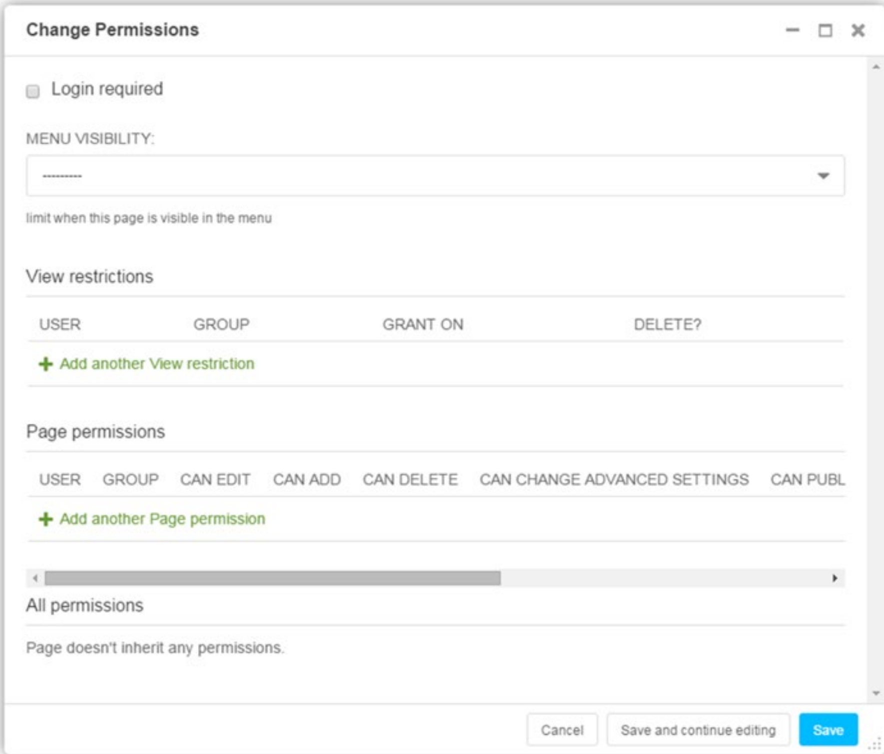


Figure 8-5. *Page permissions*

With page permissions you can:

- Require a user to be logged in to access the page.
- Hide the page in menu navigation and show only the page link in the menu for particular users.
- Restrict certain users and groups from viewing the page.
- Add or remove page editing permissions for certain users and groups.

Publishing Dates

The Publishing Dates menu option (Figure 8-6) allows you to set a date range for publication of the page. An ending date can be set for pages that should only be displayed for a limited time.

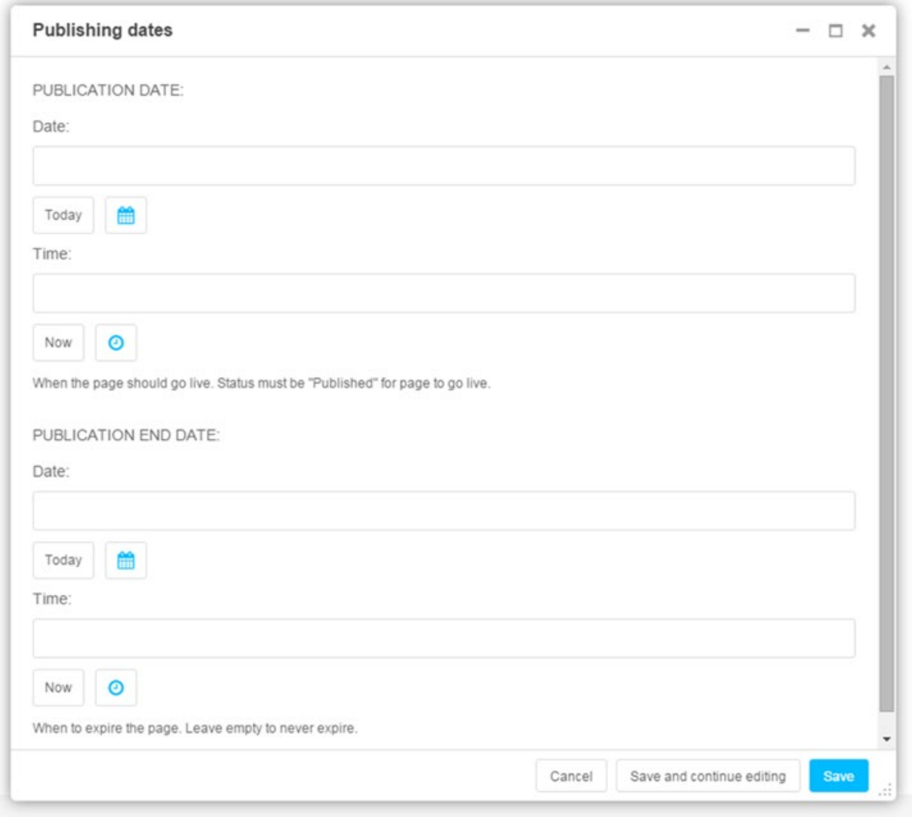


***Figure 8-6.** Page publishing dates*

Hide in Navigation

Select this option if you don't want to display the page in the top menu navigation.

Publish/Unpublish Page

Publishes or remove a previously published page from publication. Unpublished pages are visible only to editors and administrators and will no longer appear in the navigation menu for regular users.

Delete Page

Deletes a page from the CMS. Deleted pages can be recovered in the page administration interface (see later in the chapter).

Save as Page Type

Page types are a very useful tool for creating common layouts for particular types of content. For example, you may have a product review site, where each article has a picture, a summary, detailed information, and maybe a table or tabs to display specifications and recommendation. As this layout is common to all product reviews on your site, but not the site in general, you don't want to be modifying the site template.

Saving your first product review as a Page Type allows you to reuse the format for all of your product reviews. The same principle can be applied to any content on your site where you want to duplicate formatting or content. Another use for page types is an author bio at the end of each article, where you don't want to have to develop a custom plugin.

Note that the Page Type dropdown is empty when you first create your website; there are no default page types, you need to define your own.

History

django CMS retains a detailed history of the changes made to each page. The complete page history is accessed via the History menu (Figure 8-7).

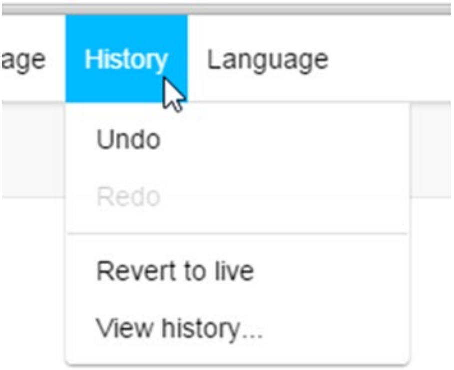


Figure 8-7. *The History menu*

The History menu is pretty simple: there is an option to undo the current changes as well as to revert to the live version, which will clear all changes back to the current published version.

■ **Caution** Revert to Live also clears any saved drafts of the page.

View History will display a complete history of a page (Figure 8-8). Selecting the revision date will give you the option of reverting back to the selected revision.

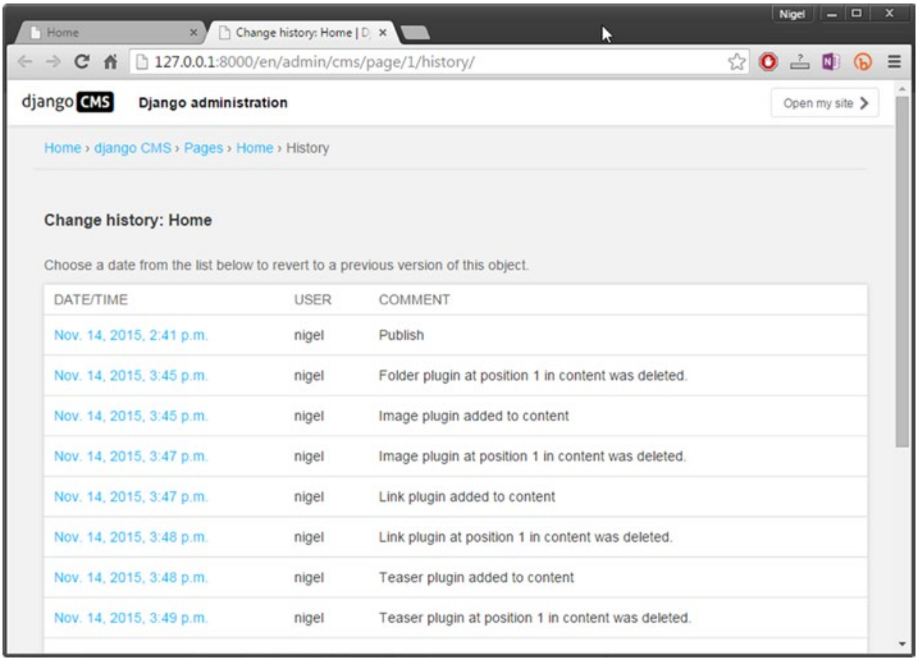


Figure 8-8. *Full page history*

Toolbar Buttons

The final functions of the django CMS toolbar are the toolbar buttons. These buttons are context-sensitive, so for an unpublished page the toolbar will look like Figure 8-9a, whereas for a previously published page it will look like Figure 8-9b. The Structure and Content buttons provide easy switching between editing the page structure and editing the content, View Published will switch from the draft version to the current published version of the page, and Publish Changes does just that: publish the current page to your site so it is visible to all users (depending on page permissions).

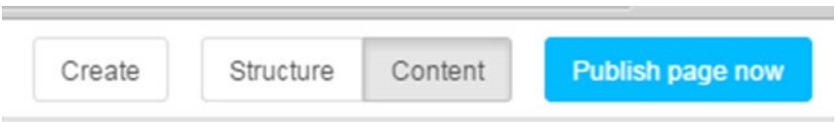


Figure 8-9a. *The toolbar buttons for an unpublished page*

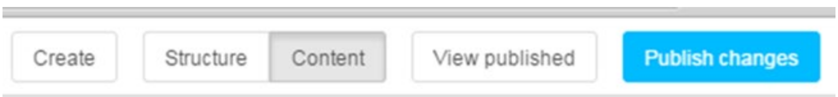


Figure 8-9b. *The toolbar buttons for an unpublished page*

django CMS automatically saves a draft of any page as soon as you start editing it. This is particularly useful for making large changes to a page, or when another user has to approve changes before they are published, as any changes will remain invisible to all but editors and administrators until they are published. This is a big improvement over CMSes like WordPress, where once a page is published, you can't save a draft edit of the page.

The Create button starts the django CMS content creation wizard. In the default installation, wizards available are for creating a Page and Sub Page. Their function is identical to that of the page creation toolbar options.

■ **Note** django CMS content creation wizards are new in version 3.2. It's possible to add your own wizards to django CMS using a similar registration process as menus and apphooks (Chapter 10). Wizards are beyond the scope of this book, but if you want to explore creating them, see:

`http://docs.django-cms.org/en/develop/how_to/wizards.html`

Adding Content to Your Site

Now that you have learned about all the different plugins available and explored the toolbar, it is time to put your new knowledge to practical use. In order for you to be able to complete the remaining chapters, your site will need some content, so in this section I have set you an exercise.

EXERCISE: CREATING YOUR SITE CONTENT

The purpose of this exercise is for you to use all of the plugins from the previous two chapters not only to learn how to use plugins in creating real content, but also because we need some content in the site so we can complete the remaining chapters in the book.

I have provided the complete sample website as a part of the source code for this book; however, it is strongly recommended that you use if only for reference, as you will learn the most by creating the pages yourself.

In this exercise, we are going to create the three most common pages found on business websites:

- A home page
- An "About Us" page
- A contact page

We are also going to create two more pages that will become blog posts a bit later on. We are going to create them now so you will have something to display on your site when we create the side menu in the next chapter.

Stage 1: Create a Home Page

1. Add a new page to your site and call it Home.
2. Switch to structure view and add an Image plugin to the content placeholder. Upload an image of your choice. For a challenge, you might want to try a Folder plugin instead and create an image slider to show several banner images, just like many popular websites.
3. Add a Text plugin to the page and enter a couple of introductory paragraphs.
4. Add a Multi Columns plugin with two columns set to 50%.
5. In the first column, add a Style plugin and then a Teaser plugin. Add some styles to format the teaser description. In the second column add another Text plugin and some text.
6. Add a final Text plugin and some closing text for the page.

Save and publish the page. I have included a sample of the home page in Figure 8-10. Please note that your page does not have to look exactly like this; the important thing is to practice using the plugins.

Stage 2: Create an About Us Page

7. Add a new page to your site and call it About.
8. Insert a single Text plugin and then insert all the content into the Text plugin. I suggest you add another image and a video and maybe a table—I'll leave the details up to your imagination.

Save and Publish the page. Figure 8-11 shows what this page might look like.

Stage 3: Create the Contact Page

9. Add a new page to your site and call it Contact
10. Add a Form plugin to the content placeholder.
11. At a minimum, your form should contain:
    - A name field of type text field
    - An email field of type email
    - A message field of type textarea

If you want to create a more advanced form, you could add radio buttons or a combo box to ask the site visitor where they first heard about your company and a checkbox to give the visitor the opportunity to email the form.

Save and publish the page. Figure 8-12 illustrates what your contact form could look like. As you can see, it is very basic and the formatting is rudimentary. As a further exercise, add a Style plugin and apply some styles to the form to make it look more professional. Remember that you will have to drag the Form plugin inside the Style plugin in your Structure editor to apply the new styles to the form.

Stage 4: Create the Blog Posts

Using some or all of the plugins, create two additional pages called “blogpost1” and “blogpost2.” Then select Page ➤ Hide in Navigation so these pages don’t show in the top menu, and publish each page. We will be hooking these pages to the side menu in the next chapter and later, once you have learned how to extend the page model, we will create a page category called “blog” in order to show them and any subsequent blog posts on our side menu as well as on the Latest Posts page.



Figure 8-10. *A sample home page*

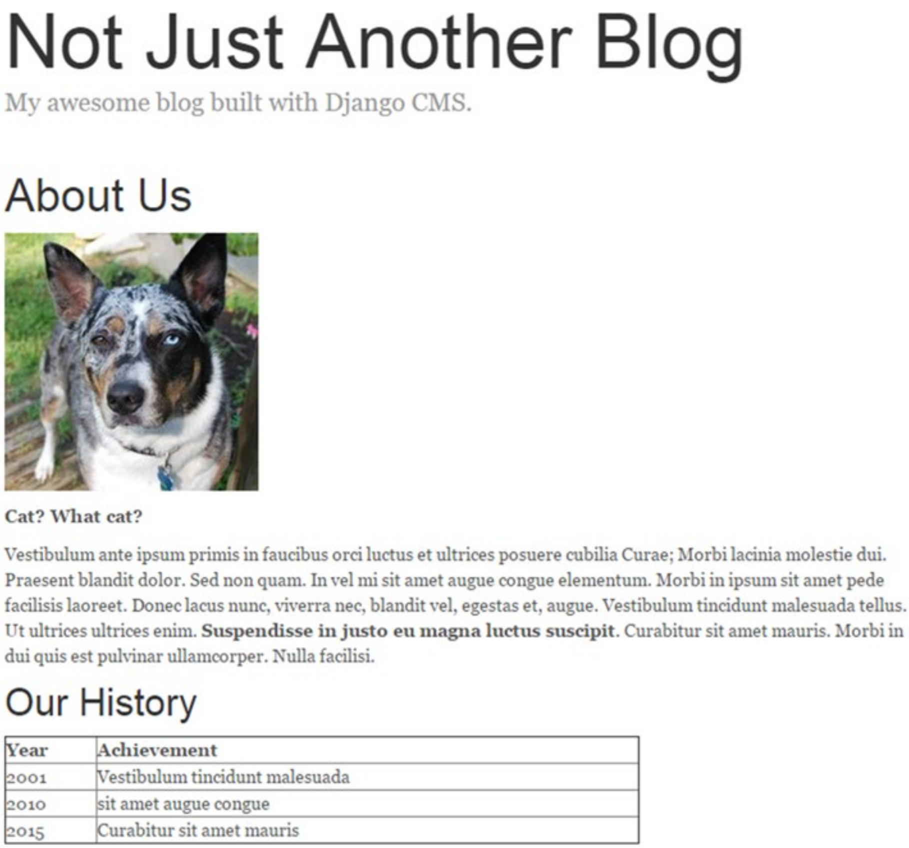


Figure 8-11. *A sample about page*

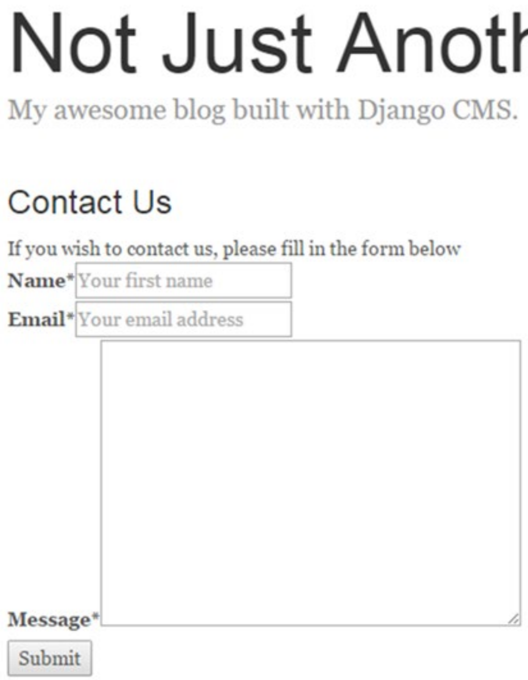


Figure 8-12. *A sample contact page*

That's it for content creation for now. Next we will explore how you administer a django CMS website.

django CMS Administration

The django CMS administration interface is accessed through the root menu (Figure 8-1) by choosing example.com ➤ Administration. The default administration dashboard is shown in Figure 8-13. Your administration dashboard may look a bit different, depending on which third-party plugins you have installed.

If you have used Django before, this should be familiar to you: the standard Django Authentication and Authorization and Sites panels are displayed, as well as a custom panel for django CMS functions. The three most common administrative tasks performed in the admin interface are:

1. Adding, editing and deleting pages
2. Changing user permission settings
3. Modifying user and group authentication and authorization settings

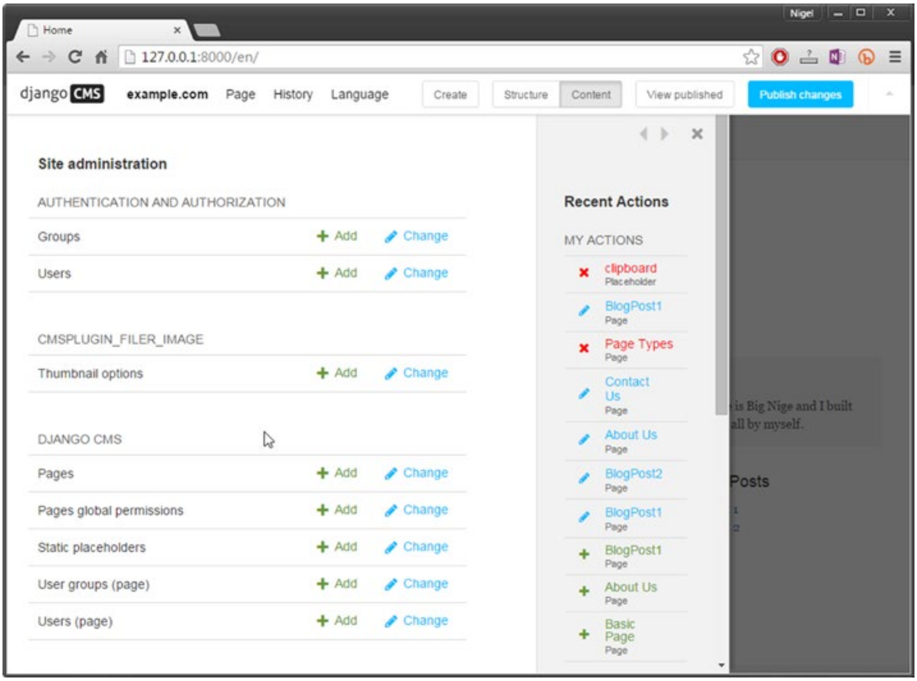


Figure 8-13. *Default administration dashboard (shown maximized)*

■ **Tip** If you want to change your root menu title from `example.com` to something like `My Blog`, go to Sites , select the `example.com` link, and change Display Name: to a new title. When you refresh the page, the toolbar root will have changed from example.com to your new title.

Page Administration

The django CMS page administration interface is shown in Figure 8-14. It can be accessed either from the administration dashboard under django CMS ➤ Pages or via the toolbar root (`example.com`) ➤ Pages. Figure 8-14 shows what your page admin screen will look like after you have completed the previous exercise. If you have jumped ahead, the page list will be blank.

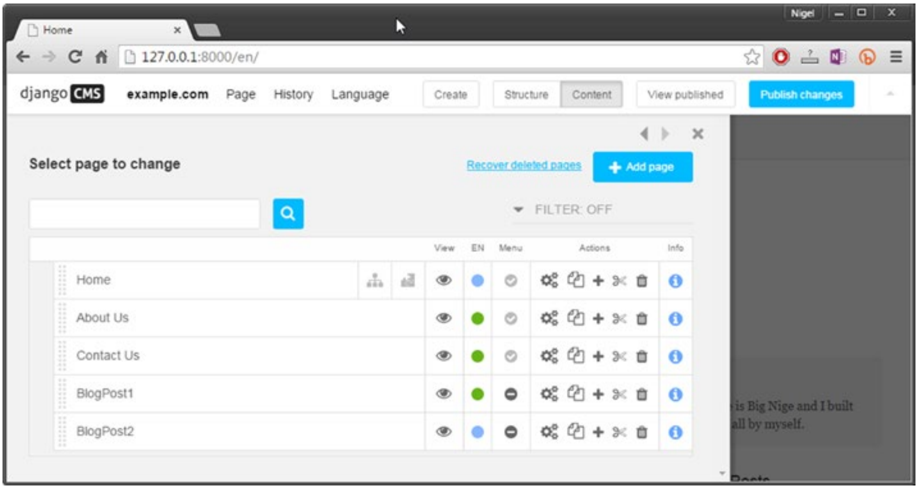


Figure 8-14. *django CMS page administration*

A number of page management functions can be performed from the page administration interface:

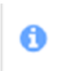
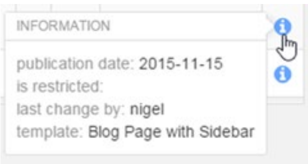
- Add a new page and recover deleted pages via the buttons in the top right.
- Reorder pages in the menu hierarchy by selecting the drag handle ( ) and dragging the page to the desired location.
- The toolbar on the right of the page listing provides some handy page management tools. Table 8-1 details each of the available tools.

Table 8-1. *Page Administration Page Tools*

Tool	Function
	**Root icon.** Displayed if the page is a root page.
	**View page.** Select to make this page the current page.
	**Published icon.** Green if the page has been published, blue if the page has unpublished changes, and gray if it's unpublished. Can be toggled by selecting the icon.

(*continued*)

Table 8-1. (*continued*)

Tool	Function
	**In Menu icon.** Checked if page is in top navigation menu. This setting can be toggled directly by selecting the icon.
	**Edit page properties.** Edits the page settings. You can also Shift-select to access advanced settings. Note that you can't edit the page contents or structure from the admin, just the settings.
	**Copy page.** When it is selected, the insert inside ( ) icon will appear next to each page in your page list. Select the page you want to insert the page under. If you want to copy the page to the root level, first insert and then use the drag handle ( ) to move the page to the desired position in the menu.
	**Add child page to current page.**
	**Cut page.** When it is selected, the insert inside ( ) icon will appear next to each page in your page list; similar to the copy function. Select the page you want to insert the page under. If you want to insert the page at root level, it's better to use the drag handle ( ) to move the page to the desired position in the menu, rather than cut and reinsert.
	**Delete the page.** Can be recovered by selecting "Recover deleted pages" button at top right.
	**Page information iconn** Hover for basic page information. Example: 

User Management

The django CMS user administration is shown in Figure 8-15. It can be accessed either from the administration dashboard under django CMS ➤ Users, or via the toolbar root (example.com) ➤ Users.

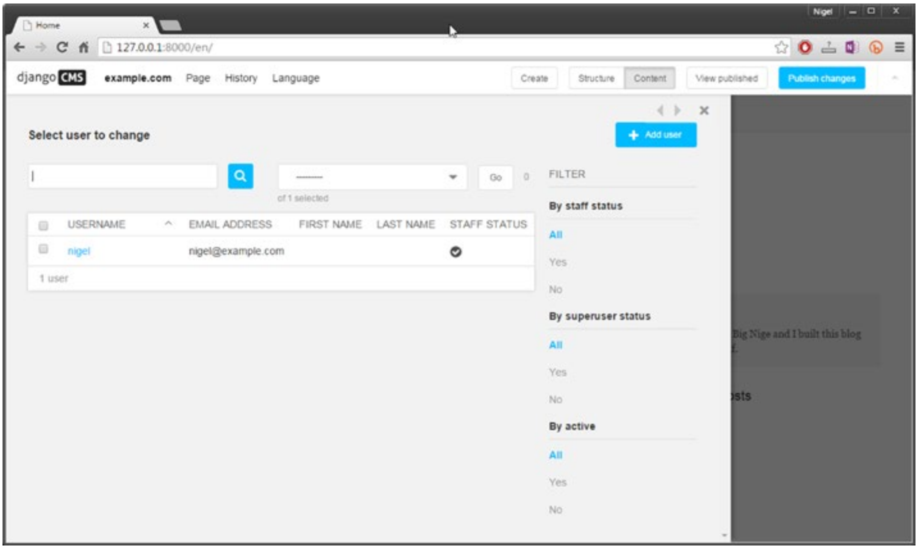


Figure 8-15. *The User admin interface*

django CMS provides a number of simple and straightforward tools to manage users:

- Change users' username, password, email and first and last name.
- Mark a user as active or inactive.
- Modify, add, and remove user permissions and groups (see next section).
- Check a user's history, signup date, and last sign-in date.

To modify a user's information, select their username and an editing window will open.

Authorization and User Permissions

Before we move on to creating and modifying user permissions, we need to understand the underlying authentication model on which they are built: the user model and assigning permissions.

User Model

The django CMS permission system is built upon Django's user authentication system, with some special permissions added specifically to simplify the management of a CMS.

Django implements a user-based model for authentication, with the most basic element being the *user object*.

The user object receives three basic permissions:

1. The ability to **add** an object
2. The ability to **change** an object
3. The ability to **delete** an object

Django automatically creates all three of these permissions for each Django model defined in your INSTALLED_APPS. To give a user permission to add, change, or delete any object in django CMS, you need to add the permissions.

To learn how this is done, go to the user administration dashboard and select a username to edit that user. Scroll down the page to the permissions panel; the fields we want to examine here are the Staff Status and Superuser Status checkboxes and the Groups and User Permissions horizontal filters.

For simple sites with only one or two administrators, it is probably OK for everyone to have full access to the site. In this case, you just need to ensure that both Staff status and Superuser status are checked for each user, and you don't need to set any more permissions; each user will have full access to the site.

If, on the other hand, you do want to control access to the site, you are going to need to set permissions. So first, let's learn how to set user permissions in Django. If you look at the User permissions filter (Figure 8-16), you will see that there is a list of permissions on the left that can be selected for each object.

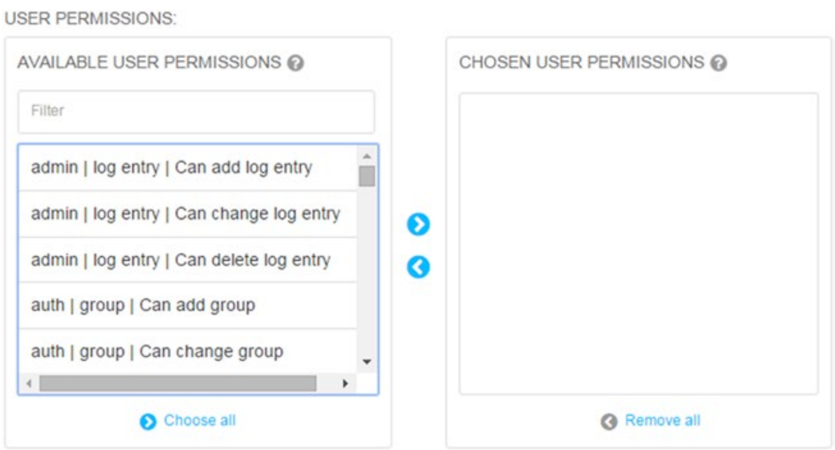


Figure 8-16. *Setting user permissions*

There are dozens of permissions in the list; however, they all follow the same format: <app> | <model> | <permission>, with three permissions for each model to allow you to assign add, change, and/or delete permissions. All of the permissions related to django CMS core are listed under the cms app (cms | …) and all of the default plugin

app permissions start with `djangocms_` (for example, `djangocms_column` for the Multi Columns plugin). If you have followed along with the other examples in the book, you will also see permissions for the `filer` and `easythumbnails` plugins.

Groups

Django groups (Figure 8-17) are a generic way of creating categories that can apply group permissions to individual users. A user assigned to a group will inherit all of the permissions assigned to that group. Users can belong to multiple groups.

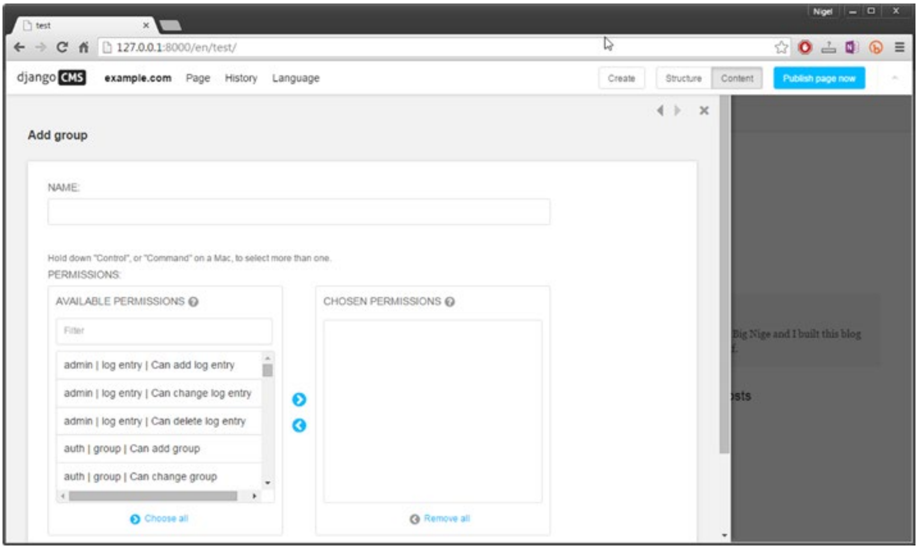


Figure 8-17. *Django user groups*

For a CMS, groups provide the obvious advantage of being able to define a set of permissions (as for an author or an editor), that can then be applied to individual users. At this stage, you might be tempted to do just that: create author and editor groups and assign each permission individually, but django CMS comes with an extension to Django's default permissions that make this quite a bit easier.

Permissions in django CMS

django CMS extends Django's user model and provides two new models for setting permissions on django CMS pages:

- User groups (page)
- Users (page)

The Users (page) model is almost identical to the Django model, so we won't cover that again here, but the User groups (page) model (Figure 8-18) adds some useful functionality that makes setting permissions for common CMS tasks very simple.

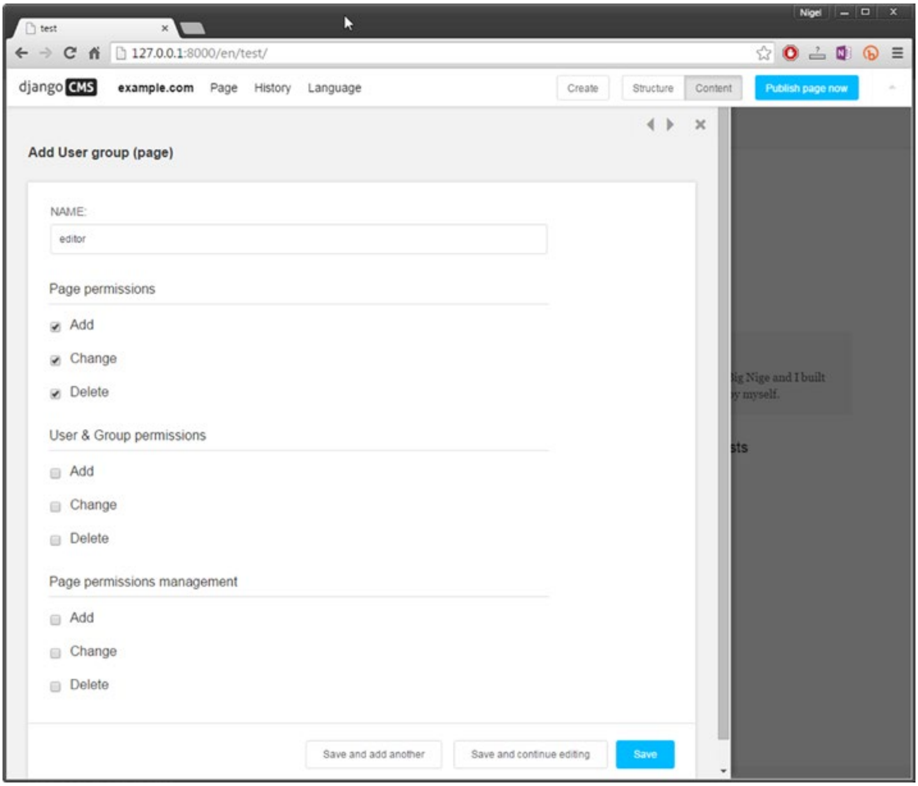


Figure 8-18. *Adding a user group*

As an example, let's create a new group called Editor. For the new group add the following permissions:

- Page Permissions: Add, change, and delete
- User & Group permissions: All unchecked
- Page permissions management: All unchecked

These permissions reflect what you would expect from a page editor: they can add change and delete pages, but can't add, change, or delete permissions for other users.

If you navigate back to the administration dashboard and open Authentication and Authorization ➤ Groups ➤ Editor, you should see something like Figure 8-19.

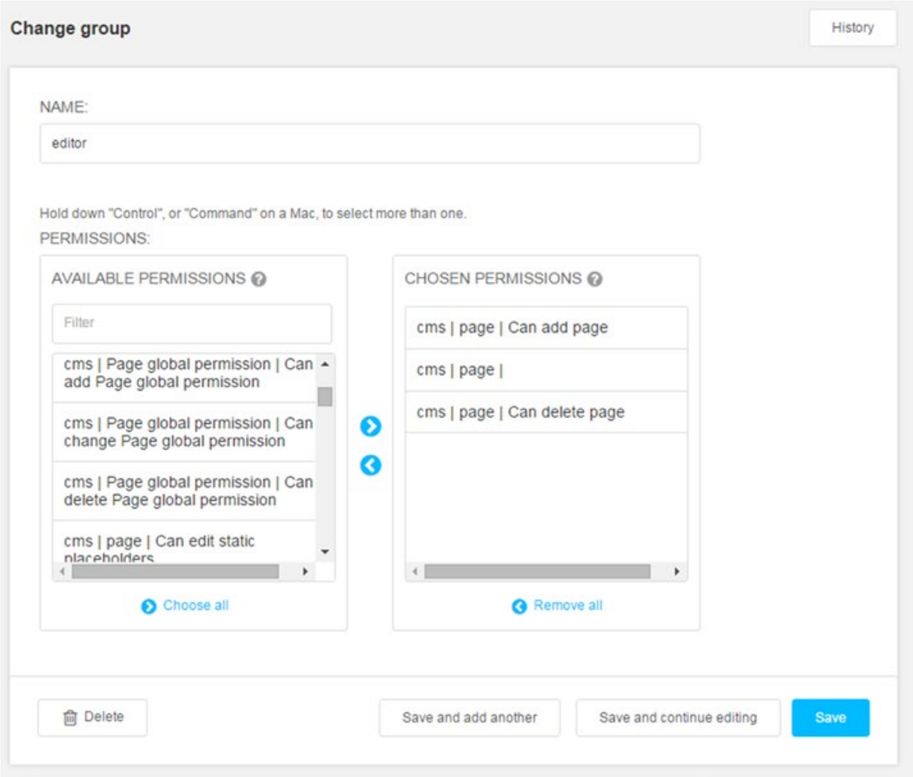


Figure 8-19. *Permissions have been added automatically to a group*

You can see here that django CMS has added a number of permissions based on your permission settings selected when you created the user group. For an exercise, create a new user and assign them to the editor group. Make sure Staff Status is checked and Superuser Status is unchecked when you save your new user. If you log out and log back in as your new user you will notice something right away: the Page menu is missing from the toolbar, and you can't edit anything!

What gives?

django CMS, like Django, has a bottom-up permission system, commonly referred to as "deny by default." To provide a full set of permissions to a user, you need to add the permissions to increase access, rather than taking them away to restrict access. This provides a more robust and secure authentication model.

When you created your user and the Editor group, your new group was granted only page editing rights, but no actual access to the plugins necessary to edit the pages. Nor were you provided with toolbar access to the page menu, as this requires global page access.

You might think it is odd not to automatically add the basic plugins like the Text plugin, as they are necessary to edit anything. I would agree, but django CMS follows the Python "explicit is better than implicit" philosophy and leaves it up to you to set the permissions.

A great example that demonstrates where this can be a good thing is a company website. Because the site administrator has the capability to restrict the plugins individually, they can allow users to add and edit text to the company website, but can restrict users from inserting links and video by not providing permission to use those plugins. Yes, they can bypass this by writing plain HTML, but this approach would be very effective against potential security issues (and potential inappropriate content!) being added by users with limited experience outside a word processor.

So, to give our new user the appropriate permissions to be able to edit pages, we need to add some more permissions. Log out as your new user and log back in as the administrator.

■ **Tip** It is possible to set the permissions incorrectly so that the toolbar disappears when you log in as another user. This is a problem because you can't log back out again without the toolbar. If you find yourself in this situation, the Django admin interface can still be accessed by appending `/admin` to your URL. Log back in as the administrator, and you will have full access again to fix whatever permissions were broken.

First, from the administration dashboard, select "Pages global permissions" and then the "Add Page global permission" button at the top. You need to set the group dropdown to your Editor group and check all the boxes except View Restricted and save the record.

Once you have done this, go back to Authentication and Authorization ➤ Groups ➤ Editor (from the administration dashboard) and select all permissions (add, change, delete) for the following:

- cms | alias plugin model
- cms | cms plugin
- cms | page
- cms | placeholder
- cms | static placeholder
- cms | title | Can add title
- cmsplugin_filer_ *
- djangocms_*
- easy_thumbnails
- filer

Once these permissions have been added to the Editor group and saved, your new user should be able to perform all the same page editing functions as your administrator, but will not have access to any of the administrative functions of the website.

Summary

That's it for this chapter. You should now have a few pages and posts added to your site and have a full understanding of how to administer users and organize content.

In the next chapter we will be making your brand new blog pages available to users via a sidebar menu, as well as looking at other navigation staples of websites: breadcrumbs and sitemaps.

CHAPTER 9

Menus and Navigation

At this stage, our complete blog website is beginning to take shape: we have built our templates, learned how all the django CMS plugins work, and used them to create some content for our site. However, we still have a way to go, as our site navigation is still very basic.

In this chapter, we are going to add some common navigation elements, starting with a custom menu containing links to your blog posts.

Later in the chapter we will add social media buttons and a sitemap that assists your second most important site visitors: search engines.

This chapter is also the first time we will be digging into writing some Django code, so if you have got this far without much knowledge of Django and Python, now is the time to refresh those skills. There are some links to great Django and Python sources in Chapter 11.

Customizing the Menu

The django CMS menu system is both flexible and extensible. So far, we have only seen how django CMS automatically adds pages to our main menu, but that is not the limit of its capability—with django CMS, you can do all of the following:

- Statically extend existing menus with custom entries.
- Create custom menus that link to your apps and plugins.
- Attach a custom menu to a page.
- Create navigation modifiers that will modify the whole menu tree (for example, to apply special formatting).

We will explore the more advanced functionalities in the next chapter, but that can be a bit complex the first time you do it, so in this chapter I will introduce you to custom menus by showing how to attach a custom Blog menu to our main menu.

■ **Note** This chapter assumes that you have already created the blog pages from the exercise in Chapter 8. If you have not done so, make sure you publish at least two new pages and select Page ➤ Hide in Navigation to hide them in the root menu.

To get started, we need to create a new page. Name your new page Blog and select the Blog Page with Sidebar template. We will be attaching the custom menu to this page.

Creating a Custom Menu

There are two options available for customizing the menu: you can statically extend the existing main menu or attach the custom menu to a specific page in the main menu.

The only difference between the two options is which django CMS class you extend when you build the menu:

- To extend the main menu statically, your custom menu class has to extend `menus.base.Menu`.
- To attach your custom menu to an existing page, your custom menu class has to extend `cms.menu_bases.CMSAttachMenu`.

This is much easier to understand with an example. Create a new file in your app directory (`/myblog/`) called `menu.py` and enter the code shown in Listing 9-1.

Listing 9-1. Custom Menu Attached to Root

```
#menu.py

from menus.base import Menu, NavigationNode
from menus.menu_pool import menu_pool
from django.utils.translation import ugettext_lazy as _

class BlogMenu(Menu):

    def get_nodes(self, request):
        nodes = []
        n1 = NavigationNode(_('Test1'), "/newpost1/", 1)
        n2 = NavigationNode(_('Test2'), "/newpost2/", 2)
        n3 = NavigationNode(_('Test3'), "/newpost3/", 3)
        nodes.append(n1)
        nodes.append(n2)
        nodes.append(n3)

        return nodes

menu_pool.register_menu(BlogMenu)
```

Let's examine this code:

- On line 1, we are importing classes `Menu` and `NavigationNode` from `menus.base`, which means the menu will be attached to the root (main menu).
- We then create our `BlogMenu` class, which extends the `Menu` class.
- The `get_nodes` function builds the menu by appending a list of menu nodes created by the `NavigationNode` class.
- Finally, we register our new custom menu with django CMS by adding it to the menu pool.

THE NAVIGATIONNODE CLASS

Central to our new custom menu class is the `NavigationNode` class, which builds our menu entries. The `get_nodes` function returns a list of `NavigationNode` instances to your application so they can be rendered in the menu. `NavigationNode` takes the following arguments:

- `Title`. The anchor text in the menu link.
- `url`. The URL of the page (internal or external) to which the menu item links.
- `Id`. Aunique ID for this menu item.
- `parent_id`. Optional; default=`None`. The ID of the parent node if this is a child node.
- `parent_namespace`. Optional; default=`None`. This is an advanced setting; see the django CMS documentation for more information.
- `Attr`. Optional; default=`None`. This is an advanced setting; see the django CMS documentation for more information.
- `Visible`. Optional; default=`True`. Set to `False` to hide this menu item.

Save your `menu.py` file and restart the development server. When you refresh your browser, you should see the items from your `PostMenu` custom menu appended to the end of the menu (Figure 9-1).

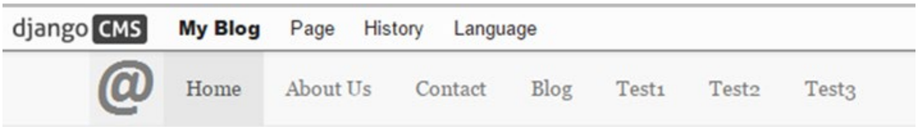


Figure 9-1. *Custom menu appended to the main menu*

A practical application for extending the root menu in this way is to add menu items that link to external applications (we will be doing this in Chapter 10) as well as other websites.

To create a menu that we can attach to a page, we need to make a few modifications to our file (Listing 9-2). I have highlighted the changes in bold.

Listing 9-2. Custom Menu Attached to Page

```
from menus.base import NavigationNode #Menu has been removed
from menus.menu_pool import menu_pool
from django.utils.translation import ugettext_lazy as _
from cms.menu_bases import CMSAttachMenu

class PostMenu(CMSAttachMenu):

    #Menu Name (shows in dropdown)
       name = _("post menu")

    def get_nodes(self, request):
        nodes = []
        n1 = NavigationNode(_('Test1'), "/newpost1/", 1)
        n2 = NavigationNode(_('Test2'), "/newpost2/", 2)
        n3 = NavigationNode(_('Test3'), "/newpost3/", 3)
        nodes.append(n1)
        nodes.append(n2)
        nodes.append(n3)

        return nodes

menu_pool.register_menu(PostMenu)
```

Once you have saved `menu.py`, you need to attach the custom menu to a page. We are going to attach it to your new blog page, so select it on the menu and then go to the django CMS toolbar and select Page ➤ Advanced Settings ➤ Attached ➤ Post. You will probably have to scroll down a bit in the Advanced Settings box to find the attached menu setting. Save your changes. Your main menu should now look like Figure 9-2.

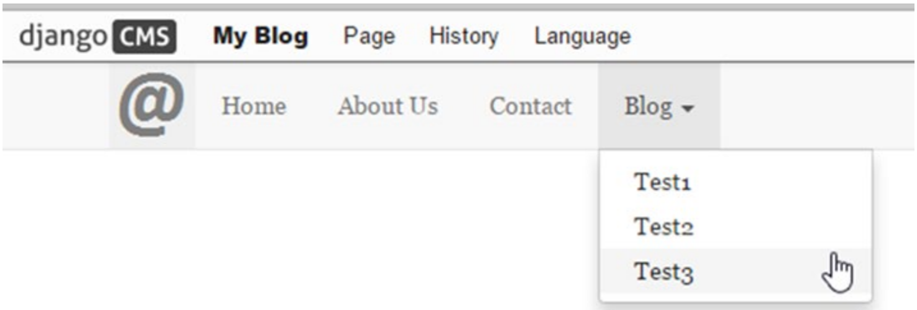


Figure 9-2. *Custom menu attached to a page*

While our new custom menu is impressive, it is not much use; the menu links are just pointing to nonexistent pages. What would make it useful is if the links pointed to our blog posts. The way we do that is pretty straightforward; we just need to replace the sample `title` and `url` values with the title and URL of our actual blog posts.

This is quite easy to do, but first we have to understand the django CMS `Page` and `Title` models a bit more. Briefly, django CMS does not store all the information about your pages in a single model; primarily it has a `Title` model that stores information about the various version of your pages (for example, language versions, metadata, and draft versions). The `Page` model stores page-related information like publication dates, security settings, and the page template.

There are many more fields in the `Title` and `Page` models than we have time to cover here; I encourage you to explore the django CMS documentation to gain a thorough understanding of these models as they are core to more advanced programming in django CMS. For the purpose of our custom menu, the only fields we are interested in are the title of our blog post and its URL. Easy?

Not quite. There are a few caveats to address, and the best way to understand what is going on is for us to digress a bit and explore the `Page` and `Title` models with the Python interactive prompt. Open a command window and make sure you have started your virtual environment. Your command prompt should look like this:

```
(myBlog) c:\Users\<your username>\MyBlogProject>
```

From your project directory, enter this:

```
python manage.py shell
```

You should be greeted by some startup messages followed by the Python interactive prompt (>>>). Enter the following code and press Enter (don't type the three dots; Python inserts them automatically):

```
>>> from cms.models import Title
>>> for t in Title.objects.all():
...     print(t.title)
```

This is straight Django (Python) code. We are looping over all the `Title` objects in the database and printing the title field from each. On my machine, this command outputs the following:

```
...
Home
Home
About Us
Contact
BlogPost1
BlogPost2
Blog
```

Obviously this is no good—looping through the titles is going to give you all the pages in the database, not just the blog posts, so we need to separate them out. You will also note that `Home` is repeated. This is because django CMS also keeps a title entry for any page that has a draft version available, so we have to exclude the draft version of any page as well.

Finding out whether a page is a draft is easy; the django CMS `Page` model includes a Boolean (True/False) attribute called `publisher_is_draft`. Because we have hidden the blog posts from the main menu, we are going to use another Boolean page attribute called `in_navigation` to extract the blog pages from the page list.

And before you complain, yes I agree, `in_navigation` is a poor way to identify the blog posts because any page that is hidden in the main menu will show up in the blog menu. Since we will not be learning how to extend the `Page` and `Title` models until the next chapter (to add a category field), consider this as an illustrative example, not a practical one!

Let's test this out with the interactive console:

```
>>> from cms.models import Title
>>> for t in Title.objects.all():
...     print(t.title, t.page.in_navigation,t.page.publisher_is_draft)
...
Home True True
Home True False
About Us True True
Contact True True
BlogPost1 False True
BlogPost2 False True
Blog True True
```

From this, we can see that the first Home page is indeed a draft and that the two blog posts are the only pages that are not in the main navigation menu (in_navigation=False). Note that we are not loading the Page model here; as page is a foreign key of the Title model, we can access the Page model from the Title model with the dot (.) operator: Title.page.<attribute>.

So now we are going to turn our newfound knowledge into some usable code. We still need to find the URL of the blog pages; we do that via the path attribute:

```
>>> from cms.models import Title
>>> for t in Title.objects.all():
...     if t.page.in_navigation==False and t.page.publisher_is_draft==False:
...             print(t.title, t.path)
...
BlogPost1 blogpost1
BlogPost2 blogpost2
>>>
```

Now that you understand how this all works, the new menu.py code (Listing 9-3) should be easy to follow (new code in bold).

Listing 9-3. The menu.py File with Modifications

```
#menu.py

from menus.base import NavigationNode #Menu has been removed
from menus.menu_pool import menu_pool
from django.utils.translation import ugettext_lazy as _
from cms.menu_bases import CMSAttachMenu
from cms.models import Title

class PostMenu(CMSAttachMenu):

    #Menu Name (shows in dropdown)
    name = _("post menu")

    def get_nodes(self, request):
        nodes = []
        for t in Title.objects.all():
    if t.page.in_navigation==False and ↩
    t.page.publisher_is_draft==False:
                n = NavigationNode(_(t.title), "/" + t.path + "/", 1)
                nodes.append(n)

        return nodes

menu_pool.register_menu(PostMenu)
```

Refresh your browser (you may need to restart the development server), and your blog custom menu should now look like Figure 9-3.

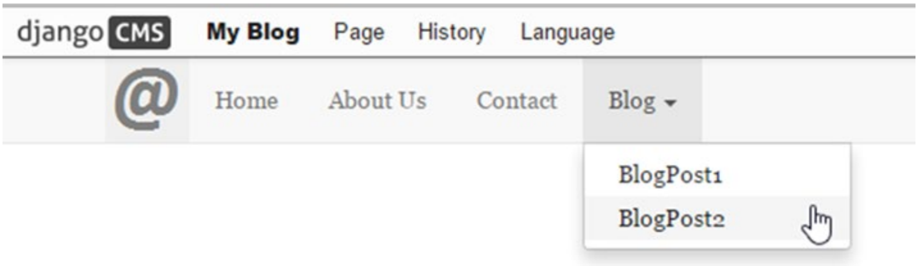


Figure 9-3. *Complete custom blog menu*

As I said earlier, this is just an example of how easy it is to build a custom menu in django CMS. I don't suggest it as a practical solution for displaying blog posts; for starters, this would create an unwieldy menu as soon as you had any more than a few blog entries.

In the next chapter, once you have learned how to extend the `Title` and `Page` models, you will be able to build much more practical custom menus. For example, once you add a Category field to each page, this menu can link to all the pages in a particular category and display the latest five posts using the Blog Roll template.

Breadcrumbs

The term *breadcrumbs* or *a breadcrumb trail* refers to a line of secondary navigation on a website that shows the user's current location in the page structure. Breadcrumbs are usually shown just above the body text of a webpage.

To add breadcrumbs to a website, django CMS provides a convenient template tag called `show_breadcrumb`. It has three optional parameters:

1. **`start_level`**. Where to start in the navigation hierarchy. Default = 0 (root).
2. **`template`**. The breadcrumb template to use. For portability, you should always create a custom template for your app. By convention, it's a file named `breadcrumb.html`, but you can name the template file anything you wish. We need to create this file for the breadcrumbs to work. In Listing 9-5 you'll see a simple `breadcrumb.html` file.
3. **`only_visible`**. Only show pages that are visible in the menu navigation. Default = `1` (true); set to `0` to show all pages.

To render a breadcrumb trail in your app, you need to add the template tag to a page template using this format:

```
{% Show_breadcrumb start_level, template, only_visible %}
```

So let's go right ahead and do that. Modify your `base.html` file with the code shown in Listing 9-4. I have included some of the existing code so you can see where to put the new code.

Listing 9-4. Modifications to `base.html` for Breadcrumbs

```
#modify base.html

. . .

<div class="blog-header">
  <h1 class="blog-title">Not Just Another Blog</h1>
 <p class="lead blog-description">My awesome blog built with django CMS.</p>
</div>

<div class="breadcrumb">
  {% show_breadcrumb 0 "breadcrumb.html" %}
</div>

{% block content %}{% endblock content %}
</div>

. . .
```

The code here is straightforward; we are displaying the breadcrumbs from the root, and we are using the `breadcrumb.html` custom template. Now we need to create the template itself (Listing 9-5). Save the file `breadcrumb.html` to `/myblog/templates`.

Also notice that I have applied the class `breadcrumb` to the div surrounding the breadcrumb template tag; `breadcrumb` is a Bootstrap class that will automatically apply some pretty formatting to our breadcrumb trail.

Listing 9-5. Our `breadcrumb.html` Template File

```
#breadcrumb.html

{% for ancestor in ancestors %}

  {% if not forloop.last %}
    <a href="{{ ancestor.get_absolute_url }}"> ↵
      {{ ancestor.get_menu_title }}</a>
    <span class="separator"> | </span>
  {% else %}
    <span class="active">{{ ancestor.get_menu_title }}</span>
  {% endif %}

{% endfor %}
```

Stepping through Listing 9-5, we can see that the Django code is looping through the navigation hierarchy and displaying each ancestor, separated by the pipe character ("|"). On the last loop through, the `if` statement fails, executing the `else` statement, which appends the current page to the breadcrumb trail. Assuming you got all the code correct, your pages should now look something like Figure 9-4.



Figure 9-4. *Our pages with the breadcrumb trail added*

Social Buttons

Social media sites like Facebook and Twitter are extremely popular for sharing all kinds of content, including favorite blog posts; so popular that Like and Share social media buttons are on almost all popular blog sites.

django CMS has a couple of third-party plugins to enable interaction with social media platforms, including Like and Share buttons; in this section, however, I am going to teach you how to embed simple social media functionality directly into your site. This is a useful skill to learn as it can be applied to the plethora of apps, other social media sites, and tools (like Google Analytics) that require you to embed a snippet of HTML and load a JavaScript library.

django CMS makes embedding social media functionality very simple; we will be using a combination of a static placeholder, some embed code from Facebook and Twitter, and a couple of JavaScript libraries. Once we are done, your blog posts will have Share buttons at the bottom just like Figure 9-5.

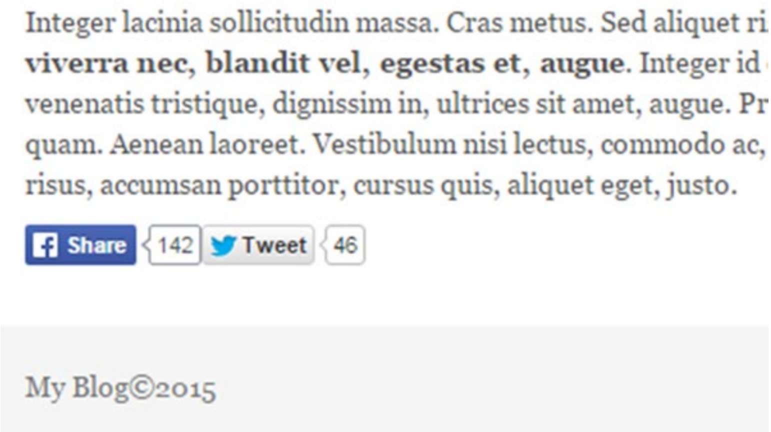


Figure 9-5. *Social media Share buttons added to blog pages*

If you remember back to Chapter 5, we have already added the following static placeholder to `blogpage.html`:

```
{% static_placeholder "social" %}
```

Select one of your blog posts from the right navigation menu and set its template to Blog Page with Sidebar (Page ➤ Templates ➤ Blog Page with Sidebar). If you change to Structure view, you will notice that you have switched to the blog template, which adds the social buttons placeholder (Figure 9-6).

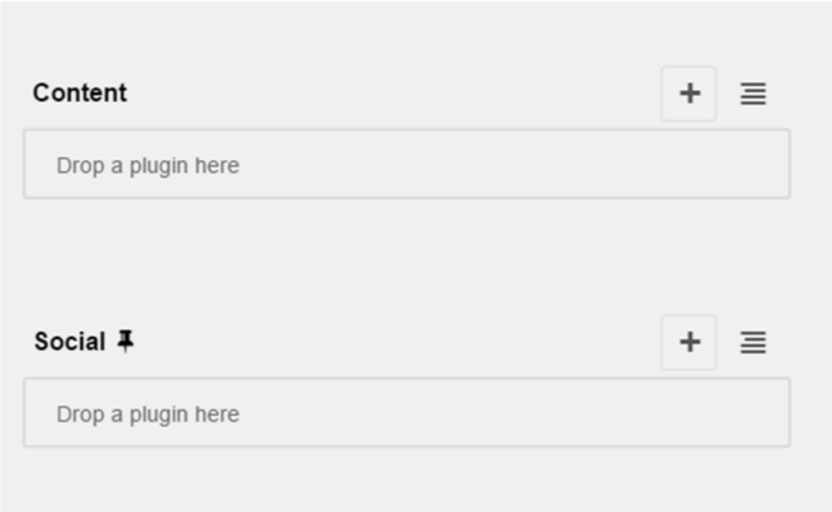


Figure 9-6. *Blog template with additional placeholders*

Add a Text plugin to the social placeholder and enter the text in Listing 9-6.

■ **Caution** Switch the Text plugin to Source mode before entering the text; otherwise it will not work.

Listing 9-6. Code to Embed Social Buttons

```
<div class="fb-share-button"
  data-layout="button_count"
  style="float: left;"> 
</div>
<div style="float: left;"><a class="twitter-share-button"
  href="https://twitter.com/intent/tweet? ↵
  text=check%20this%20out%3A%20 ">Tweet</a>
</div>
```

The first `div` embeds the Facebook share button and the second `div` embeds the Twitter share button.

Next we need to add the Facebook and Twitter JavaScript libraries. To do this, we have to modify a couple of our template files: `base.html` and `blogpage.html`.

In `base.html` we need to add an anchor for Facebook to be able to attach elements to the HTML DOM. Place the following code directly beneath your body tag:

```
<div id="fb-root"></div>
```

Then at the bottom, just above the `{% render_block "js" %}` template tag, add another template tag:

```
{% block js %}{% endblock %}
```

Because we will be adding JavaScript to the extended templates, this new block is required so that `django-sekizai` knows to add the additional scripts in our extended templates. Save `base.html` before continuing.

Now we need to modify `blogpage.html`. As this is a pretty extensive modification, I have reproduced the file in Listing 9-7, and shown the changes in bold.

Listing 9-7. `Blogpage.html` with Modifications

```
#blogpage.html

{% extends "page.html" %}
{% load cms_tags %}
{% load sekizai_tags %}
```

```
{% block title %}{% page_attribute "page_title" %}{% endblock title %}
{% block js %}
  {% addtoblock "js" %}
    <script>
      window.twttr = (function(d, s, id) {
      var js, fjs = d.getElementsByTagName(s)[0],
        t = window.twttr || {};
      if (d.getElementById(id)) return t;
      js = d.createElement(s);
      js.id = id;
      js.src = "https://platform.twitter.com/widgets.js";
      fjs.parentNode.insertBefore(js, fjs);
      t._e = [];
      t.ready = function(f) {
        t._e.push(f);
      };
      return t;
      }(document, "script", "twitter-wjs"));
    </script>
  {% endaddtoblock %}
  {% addtoblock "js" %}
    <script>(function(d, s, id) {
      var js, fjs = d.getElementsByTagName(s)[0];
      if (d.getElementById(id)) return;
      js = d.createElement(s); js.id = id;
      js.src = "//connect.facebook.net/en_US/sdk.js#xfbml=1&version=v2.5";
      fjs.parentNode.insertBefore(js, fjs);
      }(document, 'script', 'facebook-jssdk'));
    </script>
  {% endaddtoblock %}
{% endblock %}

{% block page-inner %}
<div class="blog-post">
  <h2 class="blog-post-title">Sample blog post</h2>
  <p class="blog-post-meta">October 1, 2015 by <a href="#">Nige</a></p>

    {% placeholder "content" %}

    {% static_placeholder "social" %}
    {% static_placeholder "navbuttons" %}

</div>
{% endblock page-inner %}
```

■ **Tip** The latest scripts for Twitter can be found at https://dev.twitter.com/web/javascript/loading, and for Facebook at https://developers.facebook.com/docs/plugins/share-button.

As you can see from the listing, we are using django-sekizai {% addtoblock %} tags to insert the JavaScript libraries from Twitter and Facebook. Also note that I am only loading one script per {% addtoblock %} tag. This is best practice use of django-sekizai and required for django-sekizai to be able to prevent duplicate JavaScript libraries.

If you save blogpage.html and refresh your website, you should see Facebook and Twitter share buttons at the end of each post, just as in Figure 9-5.

As I stated at the beginning of this section, this methodology is not just useful for embedding social media tools; it can be used for any tool or app that requires the addition of a HTML snippet and loading JavaScript in your pages.

Sitemaps

While there is much debate about whether a sitemap is useful, the major reason why web developers create a sitemap—Google—suggests that it's a good idea, especially for new sites. Fortunately, django CMS makes adding a sitemap so easy that my recommendation is always the same: given how little effort it takes, add a sitemap.

A sitemap is an XML file that is used by search engines to index your website. You can provide a link to your sitemap.xml file to the search engines using a tool like Google Webmaster Tools.

Django has a built-in sitemap framework: django.contrib.sitemaps. django CMS extends this framework with a new class, named CMSSitemap, which takes care of indexing titles, publication information, and location information into the sitemap so you don't have to create your own Django sitemap class.

Adding a sitemap to your site is as simple as the following steps:

1. Add django.contrib.sitemaps to your project's INSTALLED_APPS setting (should be installed by default).
2. Add from cms.sitemaps import CMSSitemap to the top of your main urls.py.
3. Add

```
url(r'^sitemap\.xml$', 'django.contrib.sitemaps.
views.sitemap', {'sitemaps':
{'cmspages': CMSSitemap}})
```

   to your urlpatterns.

Listing 9-8 shows a section of urls.py with the relevant modifications (changes in bold).

Listing 9-8. `urls.py` Sitemap Modifications

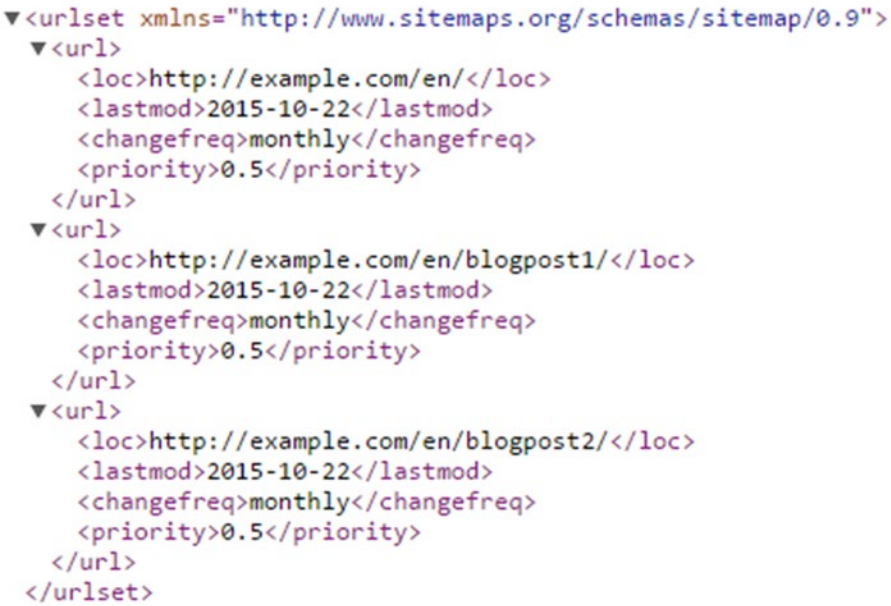
```
#urls.py (partial)
. . .

from django.contrib import admin
from django.conf import settings
from cms.sitemaps import CMSSitemap

admin.autodiscover()

urlpatterns = i18n_patterns('',
    url(r'^admin/', include(admin.site.urls)),  # NOQA
    url(r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap',
        {'sitemaps': {'cmspages': CMSSitemap}}),
    url(r'^select2/', include('django_select2.urls')),
    url(r'^', include('cms.urls')),
       url(r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap', ↵
       {'sitemaps': {'cmspages': CMSSitemap}}),
)
. . .
```

If you navigate to `http://127.0.0.1:8000/en/sitemap.xml`, you should see a sitemap similar to Figure 9-7.



```
▼<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  ▼<url>
     <loc>http://example.com/en/</loc>
     <lastmod>2015-10-22</lastmod>
     <changefreq>monthly</changefreq>
     <priority>0.5</priority>
   </url>
  ▼<url>
     <loc>http://example.com/en/blogpost1/</loc>
     <lastmod>2015-10-22</lastmod>
     <changefreq>monthly</changefreq>
     <priority>0.5</priority>
   </url>
  ▼<url>
     <loc>http://example.com/en/blogpost2/</loc>
     <lastmod>2015-10-22</lastmod>
     <changefreq>monthly</changefreq>
     <priority>0.5</priority>
   </url>
 </urlset>
```

Figure 9-7. *Snapshot of your sitemap file*

Summary

In this chapter we have covered some of the more advanced topics on menus and navigation in django CMS. We have looked at how to create a custom menu, extend the default navigation menu, create breadcrumb navigation, and add some social sharing buttons.

In the next chapter, we will dig even deeper into customizing django CMS. We will write some code to extend django CMS, and you'll see how to write your own custom apps and plugins for it.

CHAPTER 10

Extending django CMS

django CMS is highly extensible; some of this is thanks to the inherent extensibility of Django, and some to the open and pluggable design of django CMS. This is by far the number one capability of django CMS and is core to its power.

Because this is a deep and very detailed topic, this chapter can only cover some of the more important ways of extending django CMS. For those who want to dig deeper, I encourage you to explore the django CMS documentation, as well as the many online forums that discuss Django and django CMS in greater detail (see Chapter 11 for references).

First, I will show you how to extend the `Page` and `Title` models in django CMS by creating a Category field for your CMS pages. This category will be used later when building a custom menu plugin.

We will then explore how to add any kind of Django application to your website, whether a third-party application or one you have built yourself. We will be using the Polls app from the Django tutorial for this exercise, rather than building your own from scratch. I will then show you how to attach the application to your website with a django CMS apphook.

Then we will take a look at another extensible part of django CMS—the toolbar. I will introduce you to extending the toolbar by adding our Polls app so that it is editable from the front end of your website.

And finally, I will introduce custom plugins and teach you how to create and implement your own custom plugins for use in your applications, as well for sharing with other django CMS users.

Extending the Page and Title Models

In the previous chapter I used our blog pages to demonstrate how you would create a custom menu but noted that it was not a robust example, because we were using the `in_navigation` attribute to identify blog posts. What would be better is a category attribute so we could identify pages by category: a Page category for static pages on the website and a Post category for blog posts.

Out of the box, neither the `Title` model nor the `Page` model has such an attribute. Luckily for us, both the `Page` and `Title` models are extensible. The ability to extend these models is one of the powerful features of django CMS.

The process for extending each model is identical, so I will only show you how to extend the Page model here, but you'll find comments in the listings where you would change the code to extend the Title model. To add a category to each of your pages, we need to complete four steps:

1. Create the models.
2. Register the models with Django admin.
3. Create a toolbar item.
4. Add a category to each page.

Create the Models

To start, we need to build ourselves a couple of models (Listing 10-1). Save this file as models.py to your app directory (/myblog/).

Listing 10-1. The Category and CategoryExtension Models

```
from django.db import models
from cms.extensions import PageExtension #TitleExtension
from cms.extensions.extension_pool import extension_pool

class Category(models.Model):
    category = models.CharField(max_length=20)

    class Meta:
        verbose_name_plural = 'Categories'

    def __str__(self):
        return self.category

class CategoryExtension(PageExtension): #TitleExtension
    category = models.ForeignKey(Category)

extension_pool.register(CategoryExtension)
```

Stepping through this code, note the following:

- We are importing PageExtension and extension_pool from the cms.extensions class.
- The Category class is a standard Django model. The Meta class and __str__(self) function ensure that the model displays human-friendly text in the admin interface.
- Our CategoryExtension class extends the Page model. It has a single foreign key field that links to our list of categories.

- Finally, we register our extended class with the extension pool.
- To change this code so that it extends the Title model, all we have to do is replace each instance of PageExtension with TitleExtension.

Once we have created our new models, we need to add them to our application. To do this, just run these statements:

```
python manage.py makemigrations
python manage.py migrate
```

Register Models with Django Admin

Next, we need to register our new models with the admin interface, so we can edit them. Go ahead and create a new admin.py file in your app directory and enter the code shown in Listing 10-2.

Listing 10-2. Registering our Models With Django Admin

```
from django.contrib import admin
from cms.extensions import PageExtensionAdmin #TitleExtensionAdmin

from .models import Category, CategoryExtension

class CategoryAdmin(admin.ModelAdmin):
    pass

class CategoryExtensionAdmin(PageExtensionAdmin): #TitleExtensionAdmin
    pass

admin.site.register(Category, CategoryAdmin)
admin.site.register(CategoryExtension, CategoryExtensionAdmin)
```

Like the previous listing, this is standard Django code. We have created two class stubs and registered them with the Django admin application, so that we can edit the models from the back end. Once you have restarted the development server, you should have a new panel added to your admin interface (Figure 10-1).

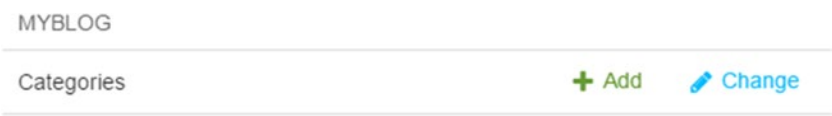


Figure 10-1. *Categories model added to admin interface*

While you are in the admin console, go ahead and add a Page and a Post category. When you are done, your categories list should look like Figure 10-2.

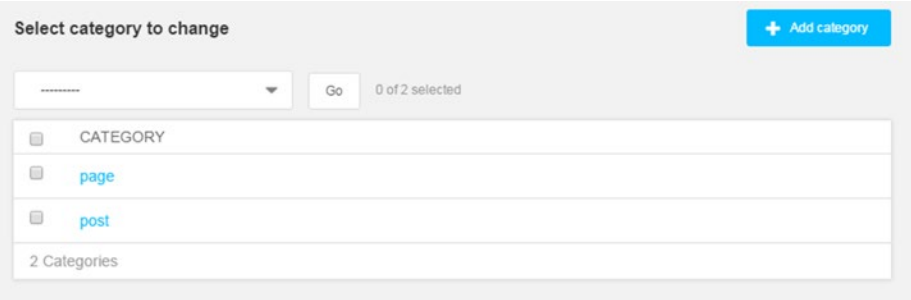


Figure 10-2. *Categories list*

Create the Toolbar Item

You will notice that our extended field does not appear in the admin interface. This is because the Page model is not directly editable in the default admin application. To attach our extended field to our pages, we need to create a toolbar item. To do this, we need to create a file called `cms_toolbar.py` (Listing 10-3) in our app directory. django CMS will look for this file when it starts and, if it exists, load the classes in the file into the toolbar.

Listing 10-3. `cms_toolbar.py`

```
from cms.toolbar_pool import toolbar_pool
from cms.extensions.toolbar import ExtensionToolbar
from django.utils.translation import ugettext_lazy as _
from .models import CategoryExtension

@toolbar_pool.register
class CategoryExtensionToolbar(ExtensionToolbar):

  model = CategoryExtension

  def populate(self):

    current_page_menu = self._setup_extension_toolbar()
```

```
if current_page_menu:
    page_extension, url = self.get_page_extension_admin()
    if url:
      current_page_menu.add_modal_item(_('Page Category'), ↵
      url=url, disabled=not self.toolbar.edit_mode)
```

Stepping through the code, note the following:

- We register our toolbar class with the `toolbar_pool.register` decorator.
- We let django CMS know that we want to use the `CategoryExtension` model for our toolbar item.
- To add our model to the toolbar, we call the `_setup_extension_toolbar()` function from `ExtensionToolbar`. This does all of the initialization and setup for our toolbar item. It returns `False` if it fails, so if there is a problem with your extended model, the function will fail gracefully by not loading your class.

Once you have entered the code, save the `cms_toolbar.py` file and restart the server. If you entered everything correctly, you will now have a new option on your Page menu (Figure 10-3).

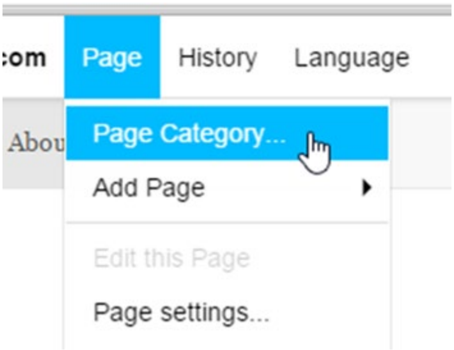


***Figure 10-3.** Page extended field added to toolbar*

Add Category to Your Pages

You are now able to add a category to each of your pages (Figure 10-4). As we will be using categories in the next section of this chapter, go ahead and assign the Blog category to each of your blog posts, and the Page category to each of the other three pages.

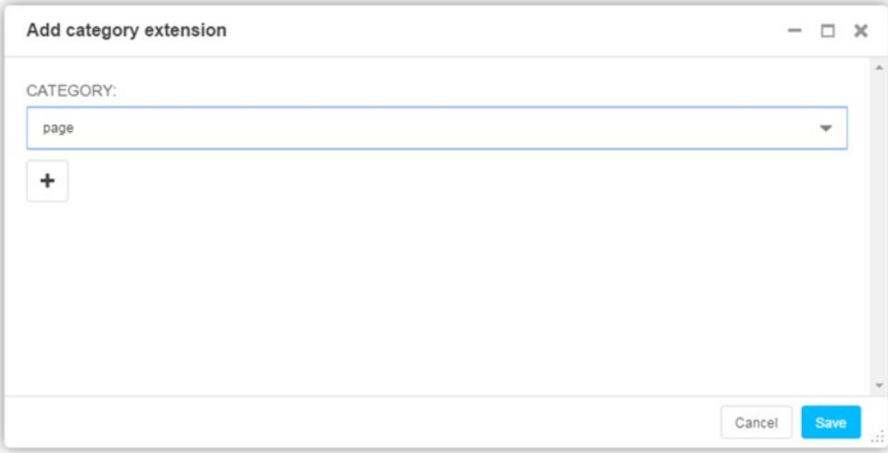


Figure 10-4. *Adding a category to a page*

Apps and Apphooks

Often you will want to extend django CMS beyond its core functions to expand the functionality of the entire website. The two ways you can extend your website are by adding or creating Django applications (Django apps) and by adding or creating django CMS plugins.

The fundamental difference between the two is that an app is a stand-alone application that is attached to your website and accessible via URL (or attached with an apphook), whereas a plugin is an object that is put into a placeholder on your page.

We will be learning about apps and apphooks in this section; plugins we will cover shortly. Adding an app to django CMS follows the same process as adding an app to Django:

1. Add a new app folder with `startapp` (or copy an existing app into your project directory).
2. Link to the app from your `urls.py` file.

For this chapter, rather than write an app from scratch, we are going to use the Polls app from the Django tutorial. To install the app in your project, follow these steps:

1. Download the Polls app from `https://github.com/big-nige/django-polls`. (Select the Download .ZIP button on the right of the page.)
2. Extract the downloaded zip file to your computer.
3. Open the extracted folder and copy the `polls` directory to your project directory.
4. Add the Polls app to your `settings.py` (Listing 10-4)
5. Add the Polls app to your `urls.py` (Listing 10-5)
6. Run `python manage.py migrate polls` from your virtual environment command prompt.

Listing 10-4. Adding the Polls App to `settings.py`

```
INSTALLED_APPS = (

 . . .

    'polls',
)
```

Listing 10-5. Adding the Polls App URLs to `urls.py`

```
urlpatterns = i18n_patterns('',

    . . .

    url(r'^polls/', include('polls.urls', namespace='polls')),
    url(r'^', include('cms.urls')),

   . . .
)
```

Once you have completed these steps, your project directory should look something like this:

```
/MyBlogProject
    . . .
    /myBlog/
    /polls/
        . . .
```

If you navigate to your administration dashboard (via the toolbar), you should see a new panel that has been added for our Polls app (Figure 10-5).



Figure 10-5. *Polls app added to the admin interface*

Go ahead and add a couple of polls. Once you have done that, you can navigate to `http://127.0.0.1:8000/polls/` to see your poll application in action.

You will notice that this is a plain page with no formatting. This is because you have not yet changed the Polls app to use your site template. To change this, edit `myBlogProject\polls\templates\polls\base.html` to match Listing 10-6.

Listing 10-6. The Modified Polls Base Template

```
{% extends 'page.html' %}

{% block content %}
    {% block polls_content %}
    {% endblock %}
{% endblock %}
```

When you refresh your `/polls/` page, your Polls app should now use your site template (Figure 10-6).



Figure 10-6. *Polls app modified to use site template*

Apphooks

While adding an application to a django CMS project is very simple, as our example Polls app demonstrates, it's still not integrated into django CMS such that it can be accessed via the site menu or modified from the front end.

An apphook allows you to attach an app to an empty django CMS page so that it's accessible via the main menu. You will learn how to create an apphook in this section.

Modifying the Polls app from the front end requires a toolbar extension, which we will cover next.

To collect and register apphooks, django CMS looks for a file called `cms_apps.py` in the root directory of your application. A basic `cms_apps.py` file is shown in Listing 10-7. Go ahead now and create this file in your `\polls\` directory.

■ **Caution** Note that the the module name is `cms_apps.py`. In previous versions of django CMS, the module was named `cms_app.py`. If you are using older Django or django CMS apps, you need to update them to the new naming convention.

Listing 10-7. `cms_apps.py`

```
from cms.app_base import CMSApp
from cms.apphook_pool import apphook_pool
from django.utils.translation import ugettext_lazy as _

class PollsApp(CMSApp):
    name = _("Poll App")
    urls = ["polls.urls"]
    app_name = "polls"

apphook_pool.register(PollsApp)
```

Stepping through this code, note the following:

- First we must import the `CMSApp` and `ApphookPool` classes (apphook_pool is an alias for the class).
- The `CMSApp` class is quite simple: our `PollsApp` subclasses it and sets three properties:
    - **Name**. Sets the apphook name. This is the name that shows in your page's advanced properties.
    - **Urls**. Links to your apps `urls.py` file.
    - **App_name**. This the name of the app as per your `INSTALLED_APPS` setting.
- Once we have created our apphook class, we add it to the apphook pool.

Save your `cms_apps.py` file and restart the development server to allow the new apphook to become available.

■ **Note** Be aware that whenever we add or remove an apphook, change the slug of a page containing an apphook, or change the slug of a page that has a descendant with an apphook, we must restart the server to reload the URL cache.

Next, you need to add a new page and navigate to Page ➤ Advanced Settings.

Scroll down until you find the Application dropdown and select Poll App from the list.

Refresh the page, and you'll find that the Polls application is now available directly from the new django CMS page (Figure 10-7).

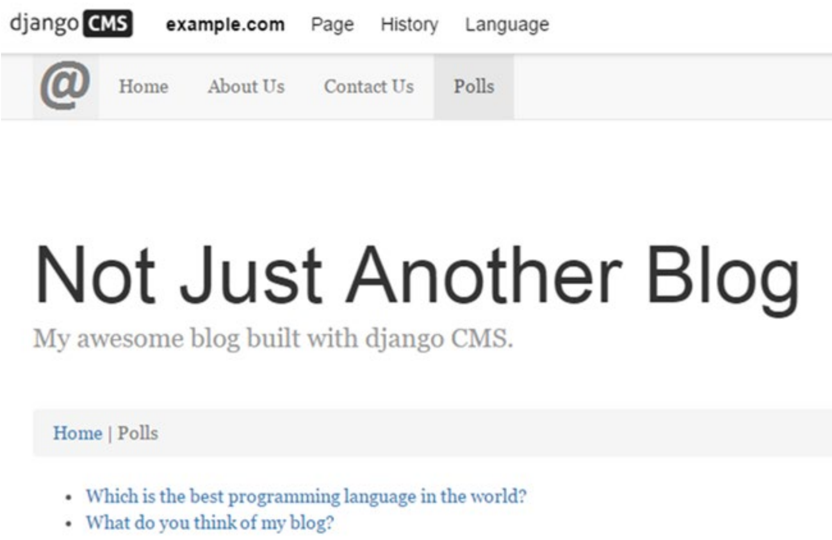


Figure 10-7. *Polls app hooked to the main menu*

■ **Note** Because the Polls app has been attached to your project with an apphook, you no longer have to refer to it in your `urls.py`. The line beginning `url(r'^polls/', . . .` can safely be removed.

To complete the integration of our Polls app in the blog website, we want to add the ability to add and modify polls from the front end. Fortunately, django CMS also allows us to extend the toolbar.

Extending the Toolbar

When we are adding apps to django CMS, it is very handy to be able to work with them from the front-end editor. django CMS provides the capability to easily extend the toolbar.

As with other extensions, django CMS will look in your application's root folder for a special file (in this case `cms_toolbar.py`) that contains a class, or classes, of type `CMSToolbar`. Each class will then be registered as a toolbar extension. For this exercise, create a new `cms_toolbar.py` file in your Polls application (Listing 10-8).

Listing 10-8. `cms_toolbar.py`

```
from django.utils.translation import ugettext_lazy as _
from cms.toolbar_pool import toolbar_pool
from cms.toolbar_base import CMSToolbar
from cms.utils.urlutils import admin_reverse
from polls.models import Poll

@toolbar_pool.register
class PollToolbar(CMSToolbar):

    watch_models = [Poll,]

    def populate(self):
        if not self.is_current_app:
            return

        menu = self.toolbar.get_or_create_menu('poll-app', _('Polls'))

        menu.add_sideframe_item(
            name=_('Poll list'),
            url=admin_reverse('polls_poll_changelist'),
        )

        menu.add_modal_item(
            name=_('Add new poll'),
            url=admin_reverse('polls_poll_add'),
        )
```

Stepping through this code, note the following:

- First we must import the `CMSToolbar` and `ToolbarPool` classes (`toolbar_pool` is an alias for the class). Note the similarity with the previous examples using apphooks. django CMS classes follow a common structure, which makes it easier to work with, and remember, each class.
- We are subclassing `CMSToolbar` within `PollToolbar` class.
- Note that this time we are using the `@register` decorator (`@toolbar_pool.register`). We could just as easily have used `toolbar_pool.register(PollToolbar)` at the end of the file.

- We set watch_models, which allows the front-end editor to redirect the user to the model instance get_absolute_url whenever an instance is created or saved through the front-end editor.
- We then define a populate() method that adds an item to the menu. The populate() method checks whether we're in a page belonging to this application, and if so, it does three things:
    - Creates a menu if one's not already there.
    - Adds a menu item to list all polls as a side frame (which will call polls.changelist() from the Django admin when selected).
    - Adds a menu item to add a new poll as a modal window (will call polls.add()from the Django admin when selected).

Save the cms_toolbar.py file and, once the development server restarts, your website should look like Figure 10-8 (note that you must be on the Polls page for this to appear).

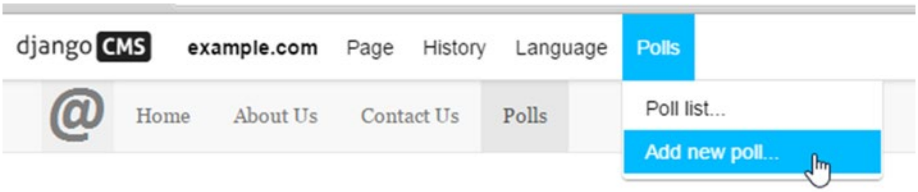


***Figure 10-8.** Polls app added to the toolbar*

Select “Add new poll” and a new window or tab will pop up (Figure 10-9), allowing you to enter a new poll.

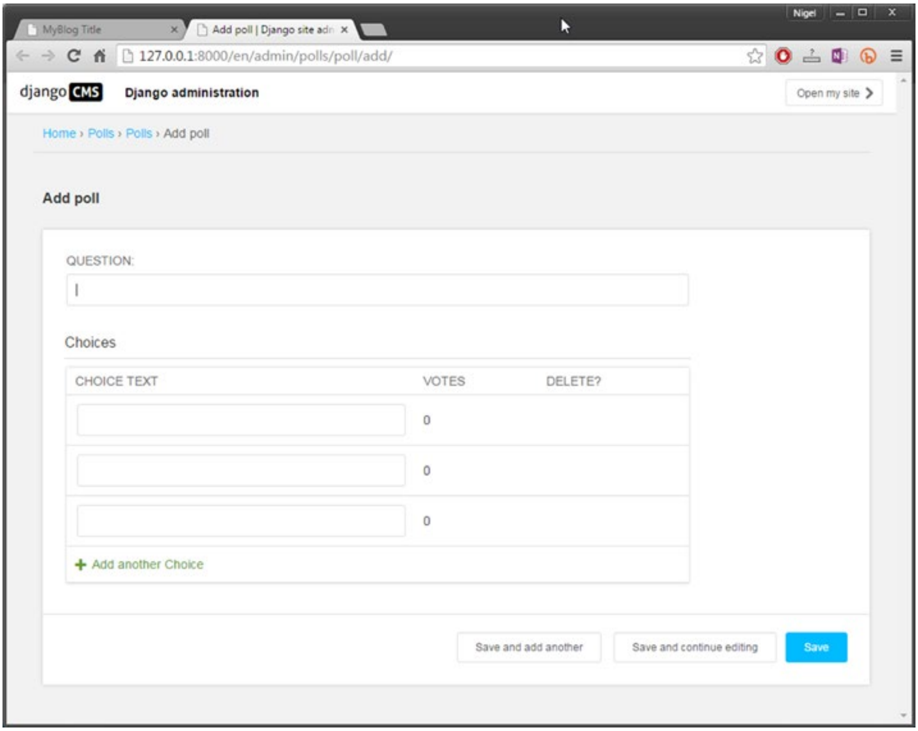


Figure 10-9. *Adding a new poll from the django CMS toolbar*

Selecting "Poll list," on the other hand, will expand the sidebar to show you the polls list (Figure 10-10).

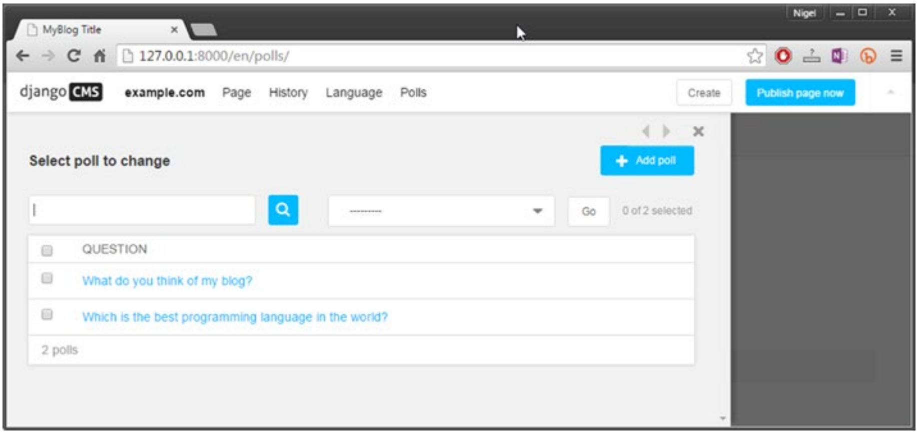


Figure 10-10. *Polls list sidebar*

■ **Note** This is a very basic example. django CMS has extensive capabilities for extending the toolbar; for example, you can add menu items to the root, but they are beyond the scope of this book. For more info, go to `http://docs.django-cms.org/en/latest/how_to/toolbar.html`.

Custom Plugins

We covered a number of the plugins available for django CMS in Chapters 6 and 7. While there are plugins available to add a variety of features to django CMS, it won't be long before you will come across a customer requirement that is not met by the available plugins. When that happens, it is time to create your own custom plugin.

It is helpful to think of a plugin in django CMS as a special kind of app that can be placed inside any django CMS placeholder. In this regard, it is no harder to create a plugin than it is to create (or add) an app and create an apphook as you did earlier in this chapter.

A django CMS plugin has three parts:

- The plugin **editor** that configures the plugin each time it's deployed. This is the plugin model and is stored in your plugin's `models.py` file.
- The plugin **publisher**. This is your plugin class that selects what is rendered to the browser. This class resides in the `cms_plugins.py` file that you must create in your plugin app's root directory.
- The plugin **template**. This is a standard django CMS template file. This file can be stored anywhere, but must be referred to by your plugin class's `render_template` attribute.

You will notice that the django CMS plugin design follows the same Model–Template–View structure as Django. In the case of plugins, the publisher is the view. To demonstrate how easy it is to create a custom plugin, we are going to start by creating a custom plugin that will display all your blog posts in a convenient sidebar navigation menu.

Sidebar Navigation

Although it is becoming less common on mobile-friendly websites, most users still expect to find a secondary menu and links (for example, to blog posts and articles) in a sidebar. Whether a left or right sidebar provides a better user experience is an argument that is unlikely to ever be resolved. I picked right simply because I like right sidebars. If you are a "leftie," feel free to experiment with the `page.html` template file. All of the code in this chapter will work regardless of which template you use (another one of those good reasons why MVC is the best design pattern).

Before You Start

Before we get started, make sure your website pages have a sidebar that looks like Figure 10-11. If it doesn't, you will need to go back to Chapter 5 and ensure you have built the templates correctly and they have been added to `settings.py`.

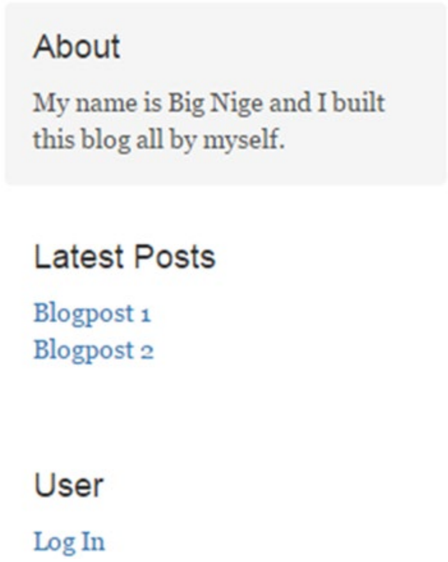


Figure 10-11. *Right sidebar navigation*

To get started with our custom plugin, we also need to create a new Django app. At your Python virtual environment command prompt, type

```
python manage.py startapp menu_plugin
```

This will create a new Django app in your `/myblog/` directory. Once you have done these two setup steps, you are ready to create your sidebar plugin. In this section we will

1. Create the publisher and template for your plugin.
2. Add a new placeholder to your page template.
3. Enhance your plugin by adding a configuration option.

Create the Plugin Publisher

As we will be starting with a very simple custom plugin, we do not need a model at this stage, so our next step is to create our publisher/view (Listing 10-9). Then, in the following section, we'll create our template (Listing 10-10).

Listing 10-9. Custom Plugin Publisher (`cms_plugins.py`)

```
from cms.plugin_base import CMSPluginBase
from cms.plugin_pool import plugin_pool
from cms.models.pluginmodel import CMSPlugin
from django.utils.translation import ugettext_lazy as _

from myblog.models import Category, CategoryExtension
from cms.models import Title

class MenuPlugin(CMSPluginBase):
    model = CMSPlugin
    name = _("Menu Plugin")
    render_template = "menu_plugin.html"
    cache = False

    def render(self, context, instance, placeholder):
      blogposts = []
      for t in Title.objects.all():
        if t.page.publisher_is_draft==False:
          try:
            if str(t.page.categoryextension.category) == 'post':
              anchor = (t.title,t.path)
              blogposts.append(anchor)
          except:
            pass

      context['blogposts'] = blogposts
      return context
plugin_pool.register_plugin(MenuPlugin)
```

Listing 10-9 looks a bit complicated, but it's actually pretty straightforward; you have already seen most of this in earlier chapters:

- You will note that this code follows a similar format to apphooks and extensions: import some custom classes, set some properties, and register the plugin with a pool so that django CMS can find it.
- The `MenuPlugin` class subclasses `CMSPluginBase` and sets some required properties:
    - **Model**. Required. This is the plugin editor that allows custom configuration of the plugin. As we are not using custom settings at this stage, we simply inherit from `CMSPlugin`.
    - **Name**. Required. The name that will be displayed in the plugin's dropdown in the front-end editor.

    - **Render template**. Required. The URL of the plugin template. As we are not providing path information, django CMS will look in `\menu_plugin\templates` for this file.
    - **Cache**. Optional. Defines whether we want to cache the plugin content.
- The `MenuPlugin` class also provides a `render()` method that passes a `context` and `instance` back to your template. Because this is the most complex part of the code, I'll explain it separately next.

The django CMS `render()` method is fundamentally the same as Django's `render()` method in that it returns an `HTTPResponse` object of the given template rendered with the given context.

In our plugin, the template is `menu_plugin.html`, and we are adding a list of tuples to the context (`blogposts`). The complexity comes from the way we need to access the extensions we made to the `Page` model at the beginning of this chapter.

To create the anchor text and URL in our side menu, we need to access the title and path of the page. If you remember from Chapter 9, `title` and `path` are both attributes of the `Title` model, but our category extension is on the `Page` model. The logic we follow is similar to the logic we used in creating the blog menu in Chapter 9:

- We step through all of the `Title` objects, but only select those that are published (`publisher_is_draft==False`).
- We then check whether the Category is `post`, and if it is, add a `(title, path)` tuple to the blogposts list.
- The `try/except` is necessary to stop Django from throwing an error when the page does not have a category assigned to it.
- We then add the blogposts list to the context for our template to render.

NOTE FOR ADVANCED USERS: CMSPLUGINBASE

The `cms.plugin_base.CMSPluginBase` class is actually a subclass of `django.contrib.admin.options.ModelAdmin`. Because `CMSPluginBase` subclasses `ModelAdmin`, several important `ModelAdmin` options are also available to CMS plugin developers. These are the most often used:

- `exclude`
- `fields`
- `fieldsets`
- `form`

- formfield_overrides
- inlines
- radio_fields
- raw_id_fields
- readonly_fields

Please note, however, that not all ModelAdmin options are effective in a CMS plugin. In particular, any options that are used exclusively by the changelist function will have no effect. For more information, see http://docs.django-cms.org/en/latest/how_to/custom_plugins.html.

Create the Plugin Template

Next, we need to create our plugin template. Create a \templates\ directory in your menu_plugin app and save Listing 10-10 to a file named menu_plugin.html.

Listing 10-10. Our Custom Plugin Template (menu_plugin.html)

```
<ol class="list-unstyled">
    {% for anchortext, link in blogposts %}
      {% with "/"|add:link|add:"/" as full_url %}
        <li><a href="{{ full_url }}">{{ anchortext }}</a></li>
      {% endwith %}
    {% empty %}
      <li>There are no published blog posts</li>
    {% endfor %}
</ol>
```

Stepping through this code, note the following:

- We are using a for loop to step over the list of tuples passed to the browser in blogposts.
- The {% with … %} tag provides a simple and effective way to concatenate a text string (in this case a forward slash /) to the beginning of a template variable. Django's standard {% value|add:'<your string>' %} tag does not work here, as you can't prepend the /.
- Each tuple in the blogposts list is used to build an HTML anchor tag to display the anchor text and link to each of your blog posts.

Once you have created your plugin template, all you need to do is add 'menu_plugin' to your INSTALLED_APPS (in settings.py) and restart the development server. Your new menu plugin will now be available to install into any of the placeholders on your pages (Figure 10-12).

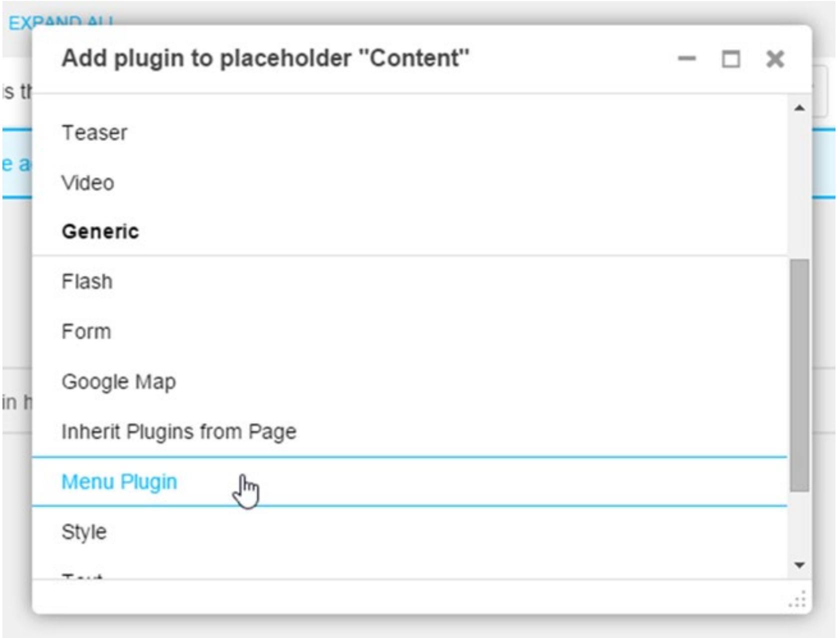


***Figure 10-12.** Custom menu plugin added to plugins*

Add Placeholder to Page Template

Of course, our custom menu isn't much use in the middle of a page; we need to be able to add it to our sidebar menu. If you remember back to Chapter 5, we created some dummy text as a placeholder in our page template (`page.html`). We now need to replace this code with a django CMS placeholder. Listing 10-11 shows the changes you need to make to `page.html`.

Listing 10-11. Modified `page.html` template

```
. . .
<h4>Latest Posts</h4>
    {% static_placeholder "posts" %}
    <ol class="list-unstyled">  #delete this line
      <li><a href="#">Blogpost 1</a></li>  #delete this line
      <li><a href="#">Blogpost 2</a></li>  #delete this line
    </ol>  #delete this line
. . .
```

Refresh your browser and switch to one of your blog posts (or any page with the right sidebar). Switch to Structure mode on the toolbar, and your page should have a Posts placeholder (Figure 10-13).

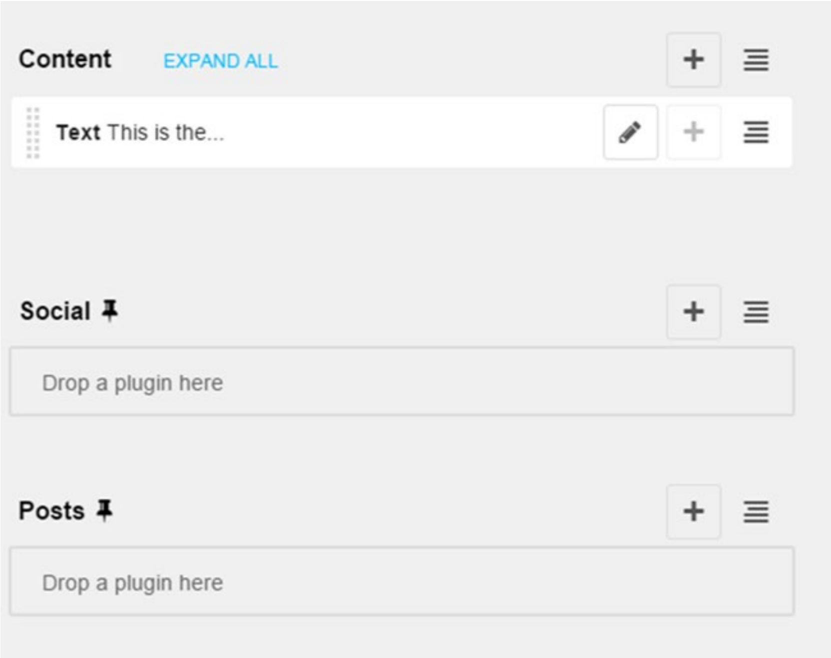


Figure 10-13. *Posts placeholder in right sidebar*

Add our custom menu plugin to this placeholder, save it, switch back to content view, and you should be greeted by a fully functional right menu (Figure 10-14) showing all your blog posts.

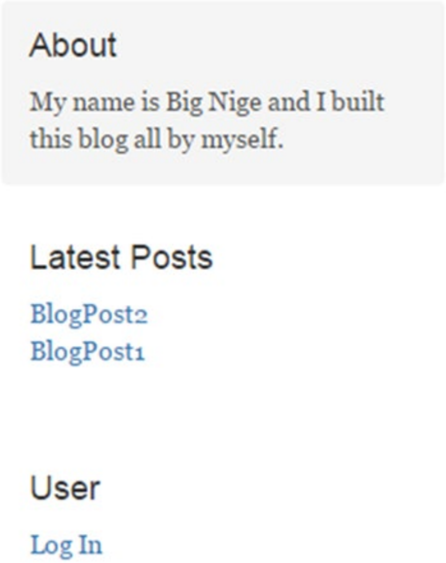


Figure 10-14. *Completed right menu with blog posts*

While our right menu is now working as designed, it does have a flaw—what if you have hundreds of blog posts? A popular answer is to create a menu that provides an archive list for years and months, rather than list every post on one page. As there is not enough scope in this book to build such a detailed model, I am going to show you a simpler solution: provide a plugin configuration option to limit the number of blog posts that show in the list.

Add Configuration Options to Your Plugin

In our first version of the menu plugin, we did not create an editor (`model.py`) for the plugin. To make our menu plugin configurable, we are going to create a custom model for our plugin so we can set how many blog posts will show in the right menu.

We will start by creating your new plugin model (Listing 10-12). Save this file to `\menu_plugin\models.py`.

Listing 10-12. Menu Plugin (`models.py`)

```
from cms.models.pluginmodel import CMSPlugin

from django.db import models

class MaxEntries(CMSPlugin):
  max_entries = models.SmallIntegerField(default=0 , ↵
  verbose_name='Maximum Entries')
```

This code should be easy to follow:

- The `MaxEntries` class subclasses the `CMSPlugin` class.
- We are setting a single configuration field (`max_entries`), which will hold the maximum number of entries allowed in the menu. We are using a default value of zero; this will correspond to unlimited entries. You will see how we implement this logic when we get to modifying the template.

We also need to make some modifications to our `cms_plugins.py` file (Listing 10-13). I have shown the changes in bold.

Listing 10-13. `cms_plugins.py` Updated to Use New Model

```
. . .

from myblog.models import Category, CategoryExtension
from cms.models import Title

from .models import MaxEntries

class MenuPlugin(CMSPluginBase):
    model = MaxEntries
    name = _("Menu Plugin")
    render_template = "menu_plugin.html"
    cache = False

    def render(self, context, instance, placeholder):
      categoryID = Category.objects.get(category='post')
      posts = CategoryExtension.objects.filter(category_id=categoryID)
      blogposts = []
      for post in posts:
        if not Title.objects.get↵
(page_id=post.extended_object_id).publisher_is_draft:
          anchor = ↵
((Title.objects.get(page_id=post.extended_object_id).title),)
          anchor += ↵
((Title.objects.get(page_id=post.extended_object_id).path),)
        blogposts.append(anchor)
      context['instance'] = instance
      context['blogposts'] = blogposts
      return context

plugin_pool.register_plugin(MenuPlugin)
```

We explored this file in detail earlier in the chapter, so I will just explain the modifications. We do the following to add our new editor to the menu plugin:

- Import the `MaxEntries` model.
- Change our plugin model from the generic `CMSPlugin` to our `MaxEntries` model.
- Pass an instance of the model back to the browser when the plugin code is called. This instance will have our `MaxEntries` field included so it can be used by the template's rendering logic.

Now that we have created the new plugin editor, we need to update the database to include the plugin model. At your Python virtual environment command prompt, type

```
python manage.py makemigrations
```

This will create a migration for your new plugin model. Then we need to migrate the database to include your new model:

```
python manage.py migrate
```

Once you have completed the migration and refreshed your browser, your plugin should now have a `MaxEntries` configuration option. Go back to your blog page and edit the menu plugin you added earlier. Your editor window should now have a Maximum Entries option (Figure 10-15).

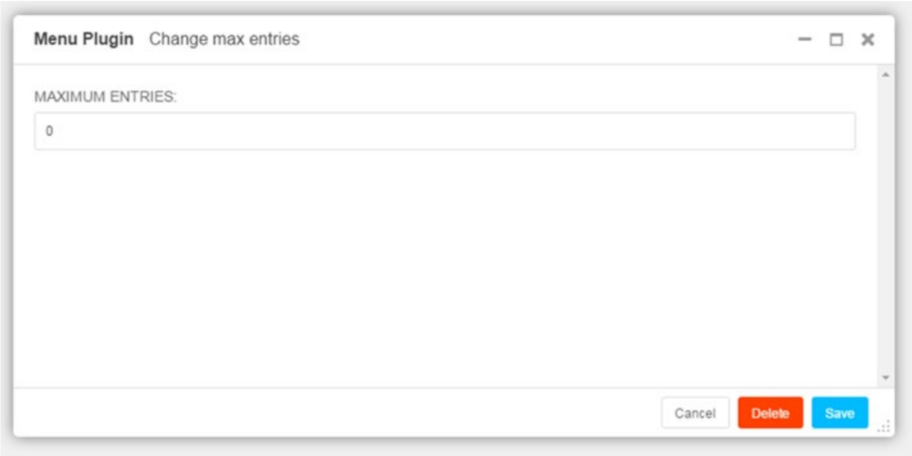


Figure 10-15. *Maximum Entries configuration option*

Because we only have a couple of blog entries at this stage, for the sake of testing our changes set this to 1 and save the plugin.

Next, we have to update our plugin template so that it will display the right number of entries in the menu (Listing 10-14). Changes are in bold.

Listing 10-14. Modified `menu_plugin.html`

```
<ol class="list-unstyled">
    {% for anchortext, link in blogposts %}
      {% with "/"|add:link|add:"/" as full_url %}
        {% if instance.max_entries == 0 %}
          <li><a href="{{ full_url }}">{{ anchortext }}</a></li>
        {% elif forloop.counter <= instance.max_entries %}
          <li><a href="{{ full_url }}">{{ anchortext }}</a></li>
        {% endif %}
      {% endwith %}
    {% empty %}
      <li>There are no published blog posts</li>
    {% endfor %}
</ol>
```

The fundamental logic in Listing 10-14 is the same as in Listing 10-10, with an additional `if … elif` statement. This new code basically says that if `max_entries` is zero (0), create anchors for all the blog posts; otherwise, only create anchors while the current count is less than or equal to `max_entries`.

Refresh your browser and, assuming you set `max_entries` to 1, you will see only the first blog post in the menu. Edit your Menu plugin and change `max_entries` to 0, save the plugin and refresh your browser, and both blog posts will reappear.

Summary

In this chapter we have explored some of the more advanced features of django CMS. You learned how to extend the `Page` and `Title` models, as well as how to add a Django app to our project and create an apphook. We used the Polls app from the Django tutorial to demonstrate how apps and apphooks are constructed and integrated into a django CMS project.

We also extended the toolbar so that we have front-end access to our new application. Finally, we learned how custom plugins are built and put our knowledge to use creating the right sidebar menu for our blog site.

This chapter is also the end of our introduction to django CMS. We have covered a great deal of ground in these 10 chapters; you should now know more than enough about django CMS to be able to build your own high-performance, professional websites.

In saying this, you should also take note that this is only the beginning of your journey with django CMS. Because this is an introductory book, I was not able to go into greater detail on many of the deeper subjects. This book has also been more of an introduction to how things work with django CMS, not necessarily what best practice is for building professional websites with django CMS.

For additional reference materials and pointers on where to go next, check out Chapter 11, which is called, appropriately, Next Steps…

CHAPTER 11

Next Steps

We have covered quite a lot of ground in this book, certainly more than enough to get you on your way to building professional-quality websites in django CMS. However, this book is only an introduction to django CMS, and there is still more to learn if you wish to develop professional programming skills in the system.

In this chapter I will provide you with tips, links, and pointers as to where you can go next in continuing your journey with django CMS. In particular, I will cover:

- Deploying django CMS
- Links to more advanced django CMS topics:
    - Understanding the menu system
    - Creating multilingual websites
    - Testing django CMS
- A list of community sites and help resources for:
    - django CMS
    - Django
    - Python

The information in this chapter is not exhaustive and, as the open source community is changing all the time, some of it may already be out of date by the time you read this. Always check the sources and follow what the community is saying; in general, the Django and django CMS communities are very active and more than willing to point newcomers in the right direction.

Deployment

Throughout this book, we have been using the Django development server to develop and test our project. While the development server is very handy while you are creating your website, it's not meant to be used in production. It's unsecured, slow, and unable to handle more than a few queries at a time.

We have also used only SQLite as our database. While SQLite is more than adequate for small sites and local deployments, it is not considered robust enough or scalable for production websites. PostgreSQL and MySQL are recommended for production sites, with PostgreSQL being the preferred option suggested by the Django Project.

With django CMS you have three deployment options (in increasing order of difficulty):

1. Deploy automatically using Aldryn.
2. Deploy on a host that supports Django.
3. Build and deploy to your own server (external or internal).

Option 3 is not recommended unless there is a very specific need; there are a number of caveats if you want a properly configured server that is capable of handling Python long-running processes. Far better to leave the back-end configuration of your Python server to an external host that knows what they are doing.

Because of the large number of configurations possible, I won't be covering the last option at all in this book. If you really want to go down this path, please refer to the Django and django CMS resources listed a little later in this chapter.

Deploy with Aldryn

Aldryn is the content management platform created by Divio (creators of django CMS) specifically to support deploying django CMS applications. The Aldryn platform provides

- automated deployment
- hosting
- shared team access
- automated installation (including dependency management) and integration of third-party add-on packages
- management via a web interface or desktop-based application

Aldryn is different from what you would expect in a standard host. Rather than just providing a tool stack and somewhere to host your Django applications, Aldryn creates a set of workflows based on the user role:

- For **content creators**, Aldryn provides a control panel for managing django CMS sites. Along with all of the functions we have covered in this book, the control panel also provides really useful functions such as a test server to run your site before it goes live and a number of preinstalled plugins and content.
- For **front-end developers**, Aldryn provides a desktop application that supports live linking between a local file repository and the production server. This allows easy development of site designs and applications without the need to run FTP servers, or to commit to external repositories like GitHub.
- For **back-end developers**, Aldryn provides command-line tools for running local builds and deploying sites.

I will not go through a deployment using Aldryn here; if you wish to explore your options with Aldryn further, see the help page (http://www.aldryn.com/en/help/) and for full documentation, see http://docs.aldryn.com/en/latest/index.html.

Deploy on a Django Host

The next best option to Aldryn, in terms of simplicity, is deploying your django CMS website on a host that explicitly supports Django. A simple web search for "Django hosts" will bring up hundreds of hosts that support Django or, for a curated list maintained by Django contributors, see https://code.djangoproject.com/wiki/DjangoFriendlyWebHosts.

At a minimum, a good host should provide all of the following:

- 1 gigabyte of dedicated RAM (the more the better)
- SSH access so you can run scripts against your server
- Periodic backups of your site and database
- Automated scripts for restarting the server periodically (to prevent lockups)

Many of the better hosts also provide "one-click" installation of a new Django application with a preconfigured database.

Installing a new django CMS project on a host that supports Django is usually no more complex that running the same scripts you ran in Chapter 2. However, there are still a number of caveats; some Django-related and some likely to be caused by the configuration on your host. Other than the host you choose, the very best resource for deploying Django on any host is the Django Project itself. Deployment options and instructions can be found here:

https://docs.djangoproject.com/en/1.8/howto/deployment/

For deployment options specifically related to installing django CMS, please refer to

http://docs.django-cms.org/en/latest/how_to/install.html

django CMS Advanced

There are a number of django CMS features that are impossible to cover in depth in an introductory book. In particular, there are three features of django CMS that you would do well to explore in depth:

1. The menu system
2. Managing multilingual sites
3. Testing

The Menu System

Although we did explore some of the important features of the django CMS menu system, there was not space to dig too deeply into this powerful tool. If you want to learn in depth how the menu system works, see

http://docs.django-cms.org/en/latest/topics/menu_system.html

There are also a number of references for the menu system in the documentation:

http://docs.django-cms.org/en/latest/reference/navigation.html

Multiple Languages

django CMS supports internationalization out of the box, with support for multilingual URLs and multiple-language versions of your pages. For a complete reference, see

http://docs.django-cms.org/en/latest/topics/i18n.html

Testing

All professional applications need to be thoroughly tested before they are deployed. Testing in django CMS uses Django's test suite:

https://docs.djangoproject.com/en/1.8/topics/testing/

Some additional constraints need to be considered when testing django CMS extensions, as they are not accessible through `urls.py`. For information on testing django CMS extensions, see

http://docs.django-cms.org/en/latest/how_to/testing.html

Getting Help

Fortunately, django CMS has the same vibrant and committed community supporting it as the open source projects it is built on: Python and Django. The following is a brief list of some leading free resources for each of the projects. The list is far from complete, but these can be considered the best places to start.

django CMS Resources

The primary django CMS reference is the documentation, which can be found at

http://docs.django-cms.org/en/latest/index.html

There are a number of community links on the documentation site under "Development & community." There are also a couple of quite active Google Groups on django CMS as well as plenty of Q&As on Stack Overflow (www.stackoverflow.com).

If you want to play with a demo of django CMS without having to download or install code, there is a good demo of django CMS at http://demo.django-cms.org.

Not long after the release of this book, I will also be publishing a number of free resources supporting the book at www.masteringdjango.com/djangocms.

Django Resources

The Django Project maintains probably the most comprehensive resource on Django:

https://docs.djangoproject.com/en/

This link will take you to the latest version of the docs. Look for links to the ever-popular Django tutorial at the top of this page. Start with the Django Project website (or search the Internet) to find other popular Django resources. These are some of the best at the time of writing:

- The Django Girls Tutorial (http://tutorial.djangogirls.org/en/index.html)
- Tango With Django (http://www.tangowithdjango.com/book17/)
- Mastering Django (www.masteringdjango.com).
- DISCLAIMER: This is my free Django resource site.
- Two Scoops of Django (www.twoscoopspress.org)

Python Resources

Python is a very mature project and as such has thousands of free (and paid) resources available. A great place to start is the list compiled by Matt Makai at Full Stack Python (http://www.fullstackpython.com/best-python-resources.html). I suggest you start with Matt's list and go from there.

If you want to learn about the differences between Python 2 and Python 3, the best place to start is https://wiki.python.org/moin/Python2orPython3.

Summary

In this chapter I have listed a few resources to get you started on your next steps with django CMS. The list is far from comprehensive, but given the changing nature of the open source landscape, it provides a starting point for what I hope will be a rewarding career in programming.

You have now reached the end of the book. I hope you enjoyed learning the material as much as I enjoyed writing it, and all the best with your future programming adventures!

Index

E

F

G, H

I, J, K

L

M

N, O

P, Q, R

S

T, U

V

W, X, Y, Z


DIVIO
Lift django CMS into the cloud
With Divio's Aldryn Cloud you'll get your website up and running in a few minutes